FICCS

Seljée, R.R.

Publication date:
1997

Document Version
Publisher's PDF, also known as Version of record

FICCS

A Fact Integrity Constraint Checking System for the Validation of Semantic Integrity Constraints after Updating Consistent Deductive Databases

R. R. Seljée

FICCS

A Fact Integrity Constraint Checking System
for the Validation of
Semantic Integrity Constraints
after
Updating
Consistent Deductive Databases

FICCS

A Fact Integrity Constraint Checking System for the Validation of Semantic Integrity Constraints after Updating Consistent Deductive Databases

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Brabant,
op gezag van de rector magnificus,
prof. dr. L.F.W. de Klerk,
in het openbaar te verdedigen
ten overstaan van een
door het college van dekanen aangewezen commissie
in de aula van de Universiteit
op

vrijdag 20 juni 1997 om 14.15 uur

door

Ronald Robertus Seljée

geboren op 12 april 1962 te Bussum.

Promotoren:

prof. dr. H.C.M. de Swart

prof. dr. P.M.E. de Bra

Preface

General Introduction Adding semantics to knowledge based systems is important for the user. The user does not want to interpret the data in the database himself, but wants to get information on the knowledge level. In order to add more semantics to knowledge based systems sophisticated data representations – such as objects, or mechanisms to group data, or object classes – are needed. Not only on the data level, but also on the inference level semantics is added to the underlying database, by production rules in rule based systems, views in relational databases or deductive rules in databases. Constraints can also add more semantics to the knowledge base. Nowadays, constraint management in knowledge based systems is becoming more important, because of the growing complexity of such system. In this thesis, the emphasis is on the way constraints can be *checked*. We look at the management of constraints in a special kind of knowledge based systems, namely deductive database systems.

In general, we not only want complete freedom in expressing all kinds of constraints on the data, but we also want the system to be responsible for checking the constraints. Deductive database management systems should offer this functionality. However, it turns out that the automated checking of integrity constraints involves a lot of complications. This thesis presents a method for checking the integrity constraints specified for a deductive database after a transaction. We are not interested in *integrity constraint enforcement*, i. e. , determining how to make an inconsistent database consistent again after a transaction, but we are only interested in the issue of *integrity constraint checking*, i. e. , determining if the database is still consistent after a transaction. In order to make integrity checking less complicated the following assumption is made:

> "*before each transaction a database is supposed to be consistent.*"

If this assumption holds, the inconsistency of the database can only be introduced by the current transaction, in which all kinds of updates, i. e. , fact updates, rule updates and constraint updates, can appear. Because of this assumption a high degree of efficiency may be reachable.

Structure of the Thesis Globally, this thesis can be divided into three parts. The first part, consisting of CHAPTER 1 and CHAPTER 2, contains an introduction to the domain of integrity checking in deductive databases. The second part, consisting of CHAPTER 3, CHAPTER 4 and CHAPTER 5, contains the theoretical results. The third part, consisting of CHAPTER 6, CHAPTER 7 and APPENDIX A, contains the practical results. The intention of writing this thesis was to partition the thesis into chapters, which would be indepent from each other as much as possible. This

resulted in a reduction of cross-referencing between chapters and of reference lists at the end of each chapter with little or no overlap.

A metaphor is used to illustrate the major concepts involved in integrity checking in deductive databases. We link integrity checking to goal keeping in soccer. A football represents a fact, a goal represents a database and a goal keeper represents one or more integrity constraints. The aim of the goal-keeper is to keep all balls from his goal, but only footballs that are visible, representing facts that lead to an inconsistency. Transparent footballs exist, which represent facts that are allowed to update the database. Therefore, in the goal-keeper's perception, he keeps every ball from the goal, although the transparent ones, being invisible to the goal-keeper, are scored. Until now, we have represented integrity checking in relational databases. However, we want to represent integrity checking in deductive databases. For this purpose, we introduce a football canon. This is a strange canon, because after a shot of a ball into this canon several balls (transparent or not) may be fired to the goal. Those shots represent the derived updates; so, the football canon represents the rules of a deductive database. Some of these shots may be with visible footballs; in other words, there may be some derived updates that lead to an inconsistency. Between every two succeeding chapters, a picture representing some part of the metaphor is given. The last picture represents the way revised inconsistency rules can be looked at in the metaphor for integrity checking in deductive databases.

Problem Description Efficiency problems arise when checking constraints automatically in knowledge based systems. The main goal of this thesis was to find causes for inefficiencies when checking integrity constraints in deductive databases and to present solutions to overcome this. In general, when considering integrity checking in deductive databases, a deductive database is supposed to be consistent with respect to a specified set of integrity constraints; it may become inconsistent when the database is updated. This assumption, which is considered to be true throughout the whole thesis, makes it possible to constrain the search for a possible inconsistency to the updates in the transaction and the part of the database that is influenced by the transaction. However, in some cases, depending on the interaction of facts, rules and constraints, this assumption cannot avoid the fact that checking the integrity constraints can still be very inefficient, when choosing the wrong search strategy for finding inconsistencies. In this thesis we study why and in which cases two well known methods for checking constraints are still inefficient. Because these methods are in some sense each others counterpart, they show different behaviour in different deductive database states, showing different aspects of inefficiencies in checking constraints in deductive databases. The language of first order logic is the most natural language for describing and studying deductive databases and their integrity constraints. Further, inefficiencies in integrity constraint checking were solved by logic programming techniques resulting in a new method for checking integrity constraints by making use of so called inconsistency rules, which detect inconsistencies in deductive databases. These inconsistency rules, which can be generated automatically from any set of rules and constraints, are deductive rules that can be incorporated in the deductive database and used as if they were deductive database rules. Hence, no meta-interpreter is needed to handle such rules.

Overview of the Thesis This thesis starts with an introduction to logic and databases. The necessary definitions, propositions and references are given in order to be able to understand the remainder of the thesis. The second chapter gives an introduction to the basic concepts on integrity checking as well as an introduction to the problems studied in this thesis; the concepts and the problems concerning integrity checking are introduced. Readers familiar with these concepts and problems can skip the first two chapters.

The main results of this thesis are presented in the next three chapters. In CHAPTER 3 efficiency problems concerning integrity checking are classified. This classification gives an overview of all kinds of redundancies that may appear in integrity checking methods. It turns out that known methods for checking integrity constraints in deductive databases do not eliminate all aspects of redundancy in integrity checking. In my opinion, this is due to the fact that the authors of those methods do not make a full inventory of all redundancy aspects, but try to improve a previously published method. By making the redundancy aspects of integrity constraint checking explicit, independently from any chosen method, it was possible to develop a new method that is optimal with respect to the classified redundancy aspects of CHAPTER 3. Although some of the redundancy aspects seem straightforward, I consider this chapter as a necessary and important basis for the realization of this thesis. The proposed method based on inconsistency rules as well as its advantages are described in CHAPTER 4. Inconsistency rules are backward chaining rules in which some forward chaining information generated from arbitrary updates, independently from the current transaction, is incorporated. Using inconsistency rules has two advantages. First, by incorporating such forward chaining information into inconsistency rules, no meta-interpreter is needed to check integrity constraints. Second, the construction of inconsistency rules can be performed completely at compile time. The possible extensions, such as augmenting the proposed method with negation, recursion and more general transactions, are described in CHAPTER 5.

The method based on inconsistency rules is usable in a system responsible for checking the semantic integrity constraints in a deductive database. We will call such a system *FICCS*, pronounced as *fix*. *FICCS* is an acronym for Fact Integrity Constraint Checking System. This name emphasizes that this subsystem of the deductive database management system is reponsible for checking the integrity constraints that are specified for base and derived *facts* in the deductive database. Being a subsystem responsible for maintaining the consistency of deductive databases, *FICCS* may be the core of the deductive database management system, because before an inconsistency can be repaired it must be detected first. Besides a subsystem like *FICCS*, subsystems responsible for maintaining the rule integrity of the rule base are needed, in order to check if rules themselves are free from redundancy and inconsistency. However, rule validation is not elaborated in this thesis, because a lot of research has already been done in that area.

CHAPTER 6 shows that this method can easily be implemented, particularly in Prolog. In CHAPTER 7 a classification of the available methods for integrity constraint checking is given. By making this classification, the disadvantages of the classified methods become clear as well. Each presented method more or less has the disadvantages belonging to the class, to which it belongs. In APPENDIX A a case study and some test results are given in order to show the practical use and

applicability of the proposed method. This case study was performed as the final project for the Master of Science Course for Knowledge Engineering at the Centre for Knowledge Engineering in Utrecht, which I accomplished besides my work as an assistent researcher at the universities of Tilburg and Eindhoven.

Project Information This thesis is the result of the research project 91P of the Co-operation Centre of Tilburg and Eindhoven Universities, which started in July 1991.

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

Contents

"Once upon a time there was a fact . . ."

Chapter 1

Introduction to Logic and Databases

A logical language is used to describe deductive databases in a natural manner: the language of first order logic.

1.1 Logic and Databases

First order logic is a very expressive language, it can be used to describe relational databases and deductive databases as well as operations on these databases. This section shows how these databases can be described by logic. More on this subject can be found in [Llo87], [Rei84] and [Ull88a].

1.1.1 Logic

The symbols used in a first-order language are (1) parentheses and brackets, (2) variables and constants, (3) predicate symbols, (4) function symbols, (5) logical connectives like $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implication), and (6) quantifiers $\forall$ (for all) and $\exists$ (there exists).

DEFINITION 1.1 A *term* is either a variable or a constant or of the form $f(t_1, t_2, \ldots, t_n)$, where f is an n-ary function symbol and $t_1, t_2, \ldots, t_n$ are terms.

DEFINITION 1.2 An expression $p(t_1, t_2, \ldots, t_n)$ is called an *atomic formula*, or an *atom*, where p is a predicate symbol of arity n and $t_1, t_2, \ldots, t_n$ are terms.

DEFINITION 1.3 Both atomic formulas and their negation are called *literals*. Atoms are called *positive literals* and negated atoms are called *negative literals* respectively.

Throughout the thesis the lower-case characters a, b, c, d, e or words consisting of lower-case characters represent constants, the upper-case characters U, V, W, X, Y, Z are used to represent variables, the lower-case characters f, g, h are used to represent function symbols, the lower-case characters p, q, r are used to represent predicate names and the lower-case characters s, t are used to represent terms.

The upper-case characters A, B, C, D, E, when appearing as arguments of a function or a predicate symbol are used to represent some constant but it is not specified which constant; so, they are used as meta-symbols and are not belonging to the first order language itself. Further, k, l, m, n, o are used for not specified fixed integers, i, j for arbitrary integers.
Some metalogical symbols are used to state that from one statement another statement can be inferred ($\Rightarrow$) ,($\Leftarrow$), and that two statements are equivalent ($\Leftrightarrow$). These symbols do not belong to the first-order language. Further, the abbreviation *iff* is used for the statement *if and only if*.

DEFINITION 1.4 The expressions allowed in a first-order language, the *well-formed formulas* (wffs) of the first-order logic, also called *first-order formulas*, are defined inductively as follows.

(i) Any literal is a wff.

(ii) If w_1 and w_2 are wffs, then $(\neg w_1)$, $(w_1 \vee w_2)$, $(w_1 \wedge w_2)$ and $(w_1 \rightarrow w_2)$ are wffs, and $\forall X[w_1(X)]$ and $\exists X[w_2(X)]$ are wffs.

(iii) The only wffs are those given by (i) and (ii).

When in the following the term "formula" is used, it stands for a well-formed formula of the first-order logic.

DEFINITION 1.5 In $\forall X[w_1(X)]$ (resp. $\exists X[w_2(X)]$) we call $w_1(X)$ (resp. $w_2(X)$) the *scope* of $\forall X$ (resp. $\exists X$).

To avoid formulas with an overload of parentheses and brackets an order of precedenceon the logical connectives and quantifiers is given:

$$\neg, \qquad \forall, \qquad \exists, \qquad \wedge, \qquad \vee, \qquad \rightarrow,$$

where each symbol in this sequence has a lower priority than the symbol to the left of it. Consequently, by $\neg w_1 \rightarrow w_2$ we mean $(\neg w_1) \rightarrow w_2$ and not $\neg(w_1 \rightarrow w_2)$; and by $w_1 \vee w_2 \rightarrow w_3$ we mean $(w_1 \vee w_2) \rightarrow w_3$ and not $w_1 \vee (w_2 \rightarrow w_3)$.

DEFINITION 1.6 Let $(w_1 \rightarrow w_2)$ be a formula. Then w_2 is called the *head* and w_1 is called the *body* of the formula. Each literal in w_2 is called a *head literal* and each literal in w_1 is called a *body literal*.

DEFINITION 1.7 Let $F : (w_1 \rightarrow w_2)$ be a formula. Let L be a body literal. Then the *side literals* of L in F, or just the side literals when from the context it is clear which body literal of which formula is meant, are all the body literals of F except this specific body literal L. When a body literal is an atom it is called a *body atom*.

DEFINITION 1.8 An occurrence of a variable X in a wff F is called *bound* if the occurrence is in a quantifier $\forall X$ or $\exists X$ or in the scope of $\forall X$ (resp. $\exists X$). An occurrence of a variable in an expression is called *free* if it is not bound.

DEFINITION 1.9 A formula is called *closed*, if it contains no free variables. Otherwise, it is called *open*.

DEFINITION 1.10 A formula is called *rectified*, if every variable is bound by at most one quantifier.

EXAMPLE 1.1 The formula $\forall X[a(X)] \rightarrow \exists X[b(X, X)]$ is not rectified.

In this thesis formulas are supposed to be rectified. This is no problem because we can rename variables, which are not rectified in order to make them rectified. For instance, in the example above a rewritten equivalent rectified formula is $\forall X a(X) \rightarrow \exists Y b(Y, Y)$

DEFINITION 1.11 A formula in which no variables occur is called a *ground* formula.

DEFINITION 1.12 A *clause* is a formula of the form $\forall X_1 \forall X_2 \cdots \forall X_n[L_1 \vee L_2 \vee \cdots \vee L_m]$, where $X_1, X_2, \ldots, X_n$ are all the variables occurring in the literals $L_1, L_2, \ldots, L_m$.

In this thesis the following logically equivalent definition of a clause is used:

$$\forall X_1 \forall X_2 \cdots \forall X_n[C_1 \wedge C_2 \wedge \cdots \wedge C_m \wedge \neg A_1 \wedge \neg A_2 \wedge \cdots \wedge \neg A_l \rightarrow H_1 \vee H_2 \vee \cdots \vee H_k]$$

where $H_1, H_2, \ldots, H_k, C_1, C_2, \ldots, C_m$ and $A_1, A_2, \ldots, A_l$ are atomic formulas. In this thesis a clause is written in the *clausal form expression*:

$$H_1 \vee H_2 \vee \cdots \vee H_k \longleftarrow C_1 \wedge C_2 \wedge \cdots \wedge C_m \wedge \neg A_1 \wedge \neg A_2 \wedge \cdots \wedge \neg A_l,$$

where the expression is supposed to be universally quantified. Because in this thesis we only consider such expressions, the logical connectives are replaced by colons where each colon represents the proper connective. Lloyd (see [Llo87]) showed that each formula can be converted into a universally quantified formula. Therefore, the restriction to those formulas may seem a limitation but in fact they are not.

DEFINITION 1.13 If it is possible to express a formula in the above clausal form it is called an *indefinite clause*. If $l = 0$ it is called a *positive indefinite clause*.
A clause is called a *normal clause* if $k = 1$.
A clause is called a *definite clause* if $l = 0$ and $k = 1$.
A clause is called a *Horn clause* if $l = 0$ and $k \leq 1$.
A clause is called a *denial* if $k = 0$.
A clause is called the *empty clause* if $k = 0$, $m = 0$ and $l = 0$; notation $\square$. The empty clause represents a contradiction.

DEFINITION 1.14 A *binding* is a mapping from a variable X to a term t which is distinct from X, denoted as X/t.

DEFINITION 1.15 A *substitution* σ for a distinct set of variables $X_1, X_2, \ldots, X_n$ is a finite set of bindings $\{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$.

The application of a substitution σ to a formula F is denoted as $F\sigma$ which results from F by replacing all variables in the substitution by the corresponding terms.

DEFINITION 1.16 A *ground substitution* for a formula F is a substitution σ for which $F\sigma$ is ground. We say that F is instantiated by σ and that $F\sigma$ is a (ground) instance of F.

Substitutions can be composed. Let σ and θ be two substitutions, then $\sigma\theta$ is also a substitution (see [Llo87] for the definition of $\sigma\theta$).

DEFINITION 1.17 A *unifier* of two formulas F and G is a substitution σ such that $F\sigma = G\sigma$. Two formulas are *unifiable* if such a substition exists.

DEFINITION 1.18 Let F be a formula and let S be a set of literals. F is called *relevant* to S and S is called *relevant* to F iff there exists a literal L in S unifiable with a literal of F. In particular, we say that F is *relevant* to L and that L is *relevant* to F.

DEFINITION 1.19 A *most general unifier* (mgu) of two formulas F and G is a unifier θ of these two formulas such that for each unifier σ there exists a substitution γ such that $\sigma = \theta\gamma$.

DEFINITION 1.20 Let F be a formula relevant to a literal A. So, there is a literal L_j in F which is unifiable with A. Let σ be a most general unifier of L_j and A. We call $F\sigma$ a *simplified instance* of F with respect to A.

1.1.2 Databases

Codd developed the relational model in the early seventies (see [Cod90]). It is still used in relational database management systems of today (see [Cod90]). However, for some purposes this model seems to be insufficient. For instance, when looking at logical databases, the relational database model is extended (with rules) to the deductive database model. Below, these two models are described.

1.1.2.1 The Relational Datamodel

The relational datamodel is described by relations. A *relation* R is a subset of $\mathcal{D}_1 \times \mathcal{D}_2 \times \ldots \times \mathcal{D}_n$, the cartesian product of so called *domains*, $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$. A relation can be represented by a table. Sometimes a relation R is referred to as table R. Here, n is called the *arity* of the relation, and $\mathcal{D}_j$ is a finite or infinite set of values for each $j = 1, 2, \ldots, n$. An element of a relation R is denoted by $\langle e_1, e_2, \ldots, e_n \rangle$ and is called a *tuple* of R. An arbitrary tuple $\langle e_1, e_2, \ldots, e_n \rangle$ for a certain n-ary relation R is abbreviated by $\bar{e}$. When expressing a tuple which can vary along a complete relation we use *tuple variables*. A tuple variable is represented by $\langle t_1, t_2, \ldots, t_n \rangle$ or $\bar{t}$, where t_j is a variable for each $j = 1, 2, \ldots, n$. Let $DI = \{R_1, R_2, \ldots, R_n\}$ be a set of relations. DI is called a *database instance* or *database state*. $DB = \{\bar{e} | \bar{e} \in R \text{ for some } R \in DI\}$ is called the *extension* of the database or, for short, database if it is clear that a relational database is involved. Each domain has a type, called *attribute*, which is a finite sequence of symbols. Let relation R be a subset of $\mathcal{D}_1 \times \mathcal{D}_2 \times \ldots \times \mathcal{D}_n$ and A_j be the attribute of domain $\mathcal{D}_j$ for each $j = 1, 2, \ldots, n$, then $R(A_1, A_2, \ldots, A_n)$ is called a *relational schema* for R. Each attribute A_j, for $j = 1, 2, \ldots, n$, from a relational schema $R(A_1, A_2, \ldots, A_n)$ can be seen as a projection, $A_j : R \rightarrow D_j$, where $A_j(\langle e_1, e_2, \ldots, e_n \rangle) = e_j$ for each tuple $\langle e_1, e_2, \ldots, e_n \rangle \in R$.

EXAMPLE 1.2 $Patient(pnr, sex)$ is a relational schema for relation $Patient$, where pnr is the attribute with domain $\{1, \ldots, 10000\}$ and sex is the attribute with domain $\{m, f\}$.

In a more practical environment n-ary relations are represented by tables with n columns, where each column has a name corresponding to the related attribute.

More details on the relational model can be found in [Cod70], [Cod90], [Dat95] and [Mai83]. For manipulating the relations in the relational model some data manipulation language is needed to do so. The most popular one is the *Structured Query Language* (SQL) which is based on *relational algebra* in which operations on relations as selection, projection, product, union, difference and all kinds of joins are defined. For an extensive introduction to this subject the references above are recommended. For a quick introduction see paragraph 27 of [Swa94].

First-order logic can describe the relational datamodel, in a very natural way. Predicates represent relations. For each relation R with n columns a corresponding n-ary predicate p_R is created for which $p_R(t_1, t_2, \ldots, t_n)$ is true if and only if tuple $\langle t_1, t_2, \ldots, t_n \rangle$ belongs to R. In the relational datamodel the type of a component is recognized by the attribute, while in the corresponding predicate the type is recognized by its position in the predicate. The relational datamodel also contains arithmetic comparison relations like $=, <, >, \geq, \leq$. Let Θ represent the set of these relations. In the logical model corresponding predicates $=, <, >, \geq, \leq$ are also available.

This distinction in relations of the relational datamodel imposes a similar distinction between predicates in the logic model.

DEFINITION 1.21 A predicate which is related to a relation in the database is called an *extensional database predicate*.

DEFINITION 1.22 A predicate which is related to an arithmetic comparison relation from Θ is called a *built-in predicate*.

DEFINITION 1.23 Let $p(c_1, c_2, \ldots, c_n)$ be a ground atom for p a n-ary predicate symbol and $c_1, c_2, \ldots, c_n$ constant symbols. If p is an extensional database predicate, then $p(c_1, c_2, \ldots, c_n)$ is called a *fact*.

DEFINITION 1.24 A *relational database* (RDB) consists of a set of facts F.

1.1.2.2 The Deductive Datamodel

A deductive database consists of facts and rules, where rules are defined in logic as follows.

DEFINITION 1.25 A *rule* is a clausal form expression.

DEFINITION 1.26 A *deductive database* (DDB) consist of a set of facts F and a set of rules R.

Three kinds of deductive databases are distinguished, for which the definition of the rules differ:

- ➢ *indefinite deductive databases*, i. e. , deductive databases where rules are clauses.
- ➢ *definite deductive databases*, i. e. , deductive databases where rules are definite clauses.
- ➢ *normal deductive databases*, i. e. , deductive databases where rules are normal clauses.

In general, a database is called *positive* if it contains no negative literals in its rules.

DEFINITION 1.27 A predicate p *directly depends on* a predicate q if a rule R exists for which p appears in a head literal of R and q appears in a body literal of R.

When p directly depends on q, we denote $q \rightarrowtail p$.

DEFINITION 1.28 A predicate p *depends on* a predicate q iff

(i) it directly depends on q, or

(ii) it directly depends on a predicate which depends on q.

When p depends on q, we denote $q \twoheadrightarrow p$.

DEFINITION 1.29 A predicate p is called *recursive* if $p \twoheadrightarrow p$. Otherwise, it is called *nonrecursive*.

DEFINITION 1.30 Two predicates p and q are called *mutually recursive* if $q \twoheadrightarrow p$ and $p \twoheadrightarrow q$.

DEFINITION 1.31 A rule R is called *recursive* if there exists a body literal of R the predicate of which is mutually recursive to the predicate of a head literal.

DEFINITION 1.32 A database D is called *recursive* if it contains a recursive rule.

When drawing dependencies between predicates of a set of clauses S in a graph, by drawing a node for each predicate in S and drawing an arc from one predicate to another if the first predicate depends on the latter, we get a *dependency graph*. A dependency graph is often represented by an *AND/OR tree*. AND/OR trees show how predicates are defined by rules. For each application of a rule with respect to its head, represented as a node in the AND/OR tree, several related AND-nodes, one for each literal in the body of the rule, are drawn. The branches to AND-nodes belonging to the rule are joined by an arc. If n rules define some predicate, then n groups of joined AND-nodes originate from that predicate. These groups are called OR-groups. OR-groups are not linked to each other. In this form a dependency graph is referred to as an *AND/OR dependency graph*. So, a database is recursive if its set of clauses has a dependency graph with one or more cycles.

EXAMPLE 1.3 Let S consisting of the clauses $p \longleftarrow q, r$, $p \longleftarrow q', r', s'$ and $q \longleftarrow s$. Then the dependency graph of S looks as in FIGURE 1.1.

DEFINITION 1.33 The variables appearing in the head of a rule are called *distinguished variables*. The variables appearing in the body of a rule that are not distinguished variables are called *nondistinguished variables*.

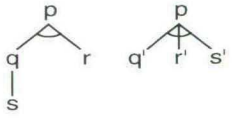


Figure 1.1: Dependency graph of EXAMPLE 1.3

An extensional database predicate is closely related to the physical part of the database. In deductive databases one has the possibility of defining new relations in terms of other existing database relations by means of rules.

DEFINITION 1.34 Let D be a deductive database. Let R be a rule in D. A predicate in the head of R is called an *intensional database predicate* if in the body of R at least one extensional database predicate occurs.

DEFINITION 1.35 Let $p(c_1, c_2, \ldots, c_n)$ be a ground atom for p a n-ary predicate symbol and $c_1, c_2, \ldots, c_n$ constant symbols. If p is an intensional database predicate, then $p(c_1, c_2, \ldots, c_n)$ is called a *derived fact*.

DEFINITION 1.36 A *database predicate* is a predicate which either is an extensional database predicate or an intensional database predicate.

DEFINITION 1.37 A literal is called a *database literal* if the literal contains a database predicate.

Other rules are used to name arithmetic expressions. Now predicates in our logic are further classified by the role they play in rules of the database.

DEFINITION 1.38 Let D be a deductive database. Let R be a rule in D. A predicate in the head of R is called an *evaluable predicate* if in the body of R there only appear system predicates or other built-in predicates, arithmetic expressions and/or evaluable predicates.

Note that an evaluable predicate is not defined in terms of any database predicate at any level of its definition.

DEFINITION 1.39 The *extensional database* consists of all facts in the database. The *intensional database* consists of all rules in the database. A *derived relation* is described by some predicate which is defined in terms of one or more predicates. A *base relation* is not defined by other relations.

DEFINITION 1.40 A deductive database is called *structured* if it does not contain any database predicate that is both extensional and intensional.

In this thesis, deductive databases are supposed to be structured. Therefore, a relation cannot be a base relation and a derived relation at the same time. For instance, the situation in the next example should not occur.

EXAMPLE 1.4 Let D be a database consisting of the following facts and rule:

$$father(fred)$$

$$mother(wilma)$$

$$husband(fred, wilma)$$

$$father(X) \longleftarrow mother(Y), husband(X, Y)$$

Here, $father(fred)$ is a fact, but can also be derived by applying the rule in the database.

As a consequence, a derived fact in a structured deductive database does not exist physically but depends purely on facts in base relations. This assumption makes the necessary definitions less complicated.

REMARK From a database which is not structured a corresponding structured database could be derived. For instance, in our previous example a new intensional predicate could be introduced, say *is_father*, which is defined as:

$$is\_father(X) \longleftarrow father(X)$$

$$is\_father(X) \longleftarrow mother(Y), husband(X, Y)$$

and replaces the rule in the example.

In real life databases one distinguishes derived relations, i. e. , views in relational databases, and base relations. Views are helpful to hide or format some parts of the database for security reasons or reasons of brevity or readability.

A database is useless if it can only store information. Its power lies in the possibility of retrieval of information. In order to retrieve information a query is posed, which is an expression representing the needed information.

DEFINITION 1.41 A *query* is a first-order formula.

However, before the answer to a query can be determined we must first get a clear notion of what information a database contains. For this reason the concept of interpretation is introduced.

1.1.3 Logical Interpretation of Databases

As before, databases are expressed by first-order formulas. Now, a semantics for these formulas is defined. Therefore, a meaning must be given to the constant, function and predicate symbols, called the interpretation of the first-order language. When a meaning is given to the syntax, one can determine the truth of the database in this interpretation. This section gives a short overview of these matters.

1.1.3.1 Interpretations and Satisfiability

We begin this section by giving the definition of *interpretation*.

DEFINITION 1.42 An *interpretation* I of a first-order language $\mathcal{L}$ consists of a non-empty set $\mathcal{D}$, called the domain of I, such that:

(i) to each constant c in $\mathcal{L}$ is assigned an element, say c^I, in $\mathcal{D}$,

(ii) to each n-ary function symbol f in $\mathcal{L}$ is assigned a mapping, say $f^I : \mathcal{D}^n \rightarrow \mathcal{D}$,

(iii) to each n-ary predicate symbol p in $\mathcal{L}$ is assigned a mapping, say $p^I : \mathcal{D}^n \rightarrow \{true, false\}$.

In each interpretation a formula gets a truth value. However, when a formula contains free variables, this truth value depends on the assignment of elements of $\mathcal{D}$ to these free variables as the next example shows.

EXAMPLE 1.5 Let $\exists Y[p(Y, X)]$ be interpreted as follows. Let $\mathcal{D}$ be the domain of the interpretation I, which represents the set of integers, and to p is assigned the predicate from $\mathcal{D} \times \mathcal{D}$ which states that the square of the first argument is equal to the second argument. When the variable X is interpreted as a negative integer the formula is not true in this interpretation with this variable assignment. However, if to X is assigned the integer 4 then the formula is true in this interpretation. Note that the variable assignment is an assignment which maps each variable symbol of $\mathcal{L}$ to an element of the domain $\mathcal{D}$. If the formula is closed, then given an interpretation the formula is either true or false, independent of the variable assignment. For instance, the closed formula $\forall X \exists Y[p(Y, X)]$ with the same interpretation as before, i. e. , the statement that all integers are equal to some squared integer, is false. So, the formula is false in this interpretation. However, one can construct interpretations in which this formula is true.

The truth value of function-free formulas in an interpretation is defined as follows:

Let p be a n-ary predicate symbol and $c_1, c_2, \ldots, c_n$ constant symbols. A fact $p(c_1, c_2, \ldots, c_n)$ is called true in interpretation I if $p^I(c_1^I, c_2^I, \ldots, c_n^I)$ holds. Otherwise, $p(c_1, c_2, \ldots, c_n)$ is called *false* in I. The truth of a more complex formula is determined by applying the following rules. Let w, w_1 and w_2 be function-free formulas, then

- $\neg w$ is true in I iff w is false in I,
- $w_1 \wedge w_2$ is true in I iff both w_1 and w_2 are true in I,
- $w_1 \vee w_2$ is true in I iff w_1 or w_2 is true in I,
- $w_1 \rightarrow w_2$ is true in I iff w_1 is false or w_2 is true in I,
- $\forall X[w(X)]$ is true in I iff for all c in $\mathcal{D}$ it holds that $w(X)$ is true in $I[X \rightarrow c]$,
- $\exists X[w(X)]$ is true in I iff for some c in $\mathcal{D}$ it holds that $w(X)$ is true in $I[X \rightarrow c]$.

The databases considered here do not contain existentially quantified formulas, because they consist only of facts and universally quantified rules. Therefore, the last rule will not be applied in determining the truth of a formula.

REMARK The truth value of closed formulas in interpretation I is independent from the variable assignment.

In this thesis databases are supposed to contain closed formulas only.

DEFINITION 1.43 Let F be a closed function-free well-formed formula in a first-order language $\mathcal{L}$. A *model* of F is an interpretation M of $\mathcal{L}$ in which F is true, notation $\models_M F$.

DEFINITION 1.44 A formula F is *satisfiable* iff there exists a model of F.

DEFINITION 1.45 A formula F is *valid* if each interpretation is a model of F.

These definitions are generalized for sets of formulas.

DEFINITION 1.46 A *model* of a set S of formulas is an interpretation M in which each formula in that set is true, notation $\models_M S$.
A set of formulas S is *satisfiable* iff there exists at least one model of S.
A set of formulas S is *valid* if each interpretation is a model of S.

DEFINITION 1.47 A formula F is a *logical consequence* of a set of formulas S if each model of S is also a model of F, notation $S \models F$. $S_1 \models S_2$ means that $S_1 \models F_1 \vee F_2 \vee \cdots \vee F_n$, where $F_1, F_2, \ldots, F_n$ are the formulas occurring in S_2.

1.1.3.2 Herbrand Interpretations

The *Herbrand universe* $\mathcal{U}_\mathcal{L}$ for a first-order language $\mathcal{L}$ is the set of all ground terms constructable from constants and function symbols occurring in $\mathcal{L}$.

Let F be a formula (resp. S be a set of formulas). Then the first-order language $\mathcal{L}_F$ (resp. $\mathcal{L}_S$) consists of all symbols appearing in F (resp. S). By the Herbrand universe $\mathcal{U}_F$ (resp. $\mathcal{U}_S$) of a formula F (resp. a set of formulas S) the Herbrand universe of $\mathcal{L}_F$ (resp. $\mathcal{L}_S$) is denoted.

DEFINITION 1.48 An *Herbrand interpretation* for a first-order language $\mathcal{L}$ is an interpretation I which has the Herbrand universe $\mathcal{U}_\mathcal{L}$ as domain and in which for every constant c, $c^I = c$.

From now on, by a Herbrand interpretation of a formula F (resp. a set of formulas S) we mean a Herbrand interpretation, where the domain is $\mathcal{U}_F$ (resp. $\mathcal{U}_S$).

DEFINITION 1.49 The *Herbrand base* $\mathcal{B}_\mathcal{L}$ for a first-order language $\mathcal{L}$ is the set of all ground atoms constructable from predicate symbols with ground terms from the Herbrand universe $\mathcal{U}_\mathcal{L}$ as arguments.

By the Herbrand base $\mathcal{B}_F$ (resp. $\mathcal{B}_S$) of a formula F (resp. a set of formulas S) we mean the Herbrand base of the first-order language $\mathcal{L}_F$ (resp. $\mathcal{L}_S$).

DEFINITION 1.50 When a Herbrand interpretation is a model of a formula F (resp. a set of formulas S), it is called a *Herbrand model* of F (resp. S).

More about Herbrand models and some examples can be found in [Llo87] and [Swa94]. Herbrand interpretations play an important role in expressing the semantics of databases, as is shown in the next section.

1.1.3.3 Herbrand Interpretations for Databases

In this thesis databases are supposed to contain closed formulas only, so, the truth value of an interpreted formula is independent of the variable assignment. Also in this thesis databases are supposed to be function-free. Consequently, databases are represented in a function-free first-order logic, i. e. , in this logic a *term* is either a constant or a variable. Therefore, the interpretation of function symbols, see (ii) in the definition of interpretation, is not relevant here. Also a database D is supposed to consist of clausal-form expressions. Now, the following definition of a Herbrand interpretation is used.

DEFINITION 1.51 Let D be a database consisting of function-free, clausal-form expressions. Let $\mathcal{U}_D$ be the Herbrand universe of D. I is a *Herbrand interpretation* of D, if

(i) to each constant c in $\mathcal{L}_D$ is assigned an element, say c^I, in $\mathcal{U}_D$, such that $c^I = c$, and

(ii) to each n-ary predicate symbol p in $\mathcal{L}_D$ is assigned a mapping, say $p^I : \mathcal{U}_D^n \rightarrow \{true, false\}$.

REMARK Because no function symbols are used, $\mathcal{U}_D$ is the set of all constants occurring in $\mathcal{L}_D$. Also when two Herbrand interpretations for a fixed set of formulas are compared, the constant assignments are equal while Herbrand interpretations may be different by the second condition for Herbrand interpretations. Note, however, that each subset S of the Herbrand base $\mathcal{B}_D$ corresponds one-to-one to a mapping from $\mathcal{U}_D^n$ to $\{true, false\}$: for each $p(e_1, e_2, \ldots, e_n) \in S$, to $p(e_1, e_2, \ldots, e_n)$ is assigned $true$ and for each $q(d_1, d_2, \ldots, d_m) \in \mathcal{B}_D/S$ to $q(d_1, d_2, \ldots, d_m)$ is assigned $false$. Therefore, we can identify a Herbrand interpretation with a subset of the Herbrand base. So, the definition of a Herbrand interpretation can be reduced to "a Herbrand interpretation of a function-free database D is a subset of its Herbrand base $\mathcal{B}_D$".

More about variable assignments and first-order languages with function symbols can be found in [Llo87], [Swa93] and [Swa94].

The effort in defining Herbrand interpretations (and models) is unnecessary, because satisfiability of a database can be defined in terms of the existence of a more general interpretation (and model).

Let M be a model of a set of clauses, say D, expressed in a first-order language $\mathcal{L}$. By definition, each formula in D is true in interpretation M. The truth of a formula is determined by the truth of its atomic components. Let $p(c_1, c_2, \ldots, c_n)$ be a ground instance of an atom appearing in D, where p and $c_1, c_2, \ldots, c_n$ are symbols in $\mathcal{L}$. Now, when in the interpretation M, $p^M(c_1^M, c_2^M, \ldots, c_n^M)$ is true, then we define $p(c_1, c_2, \ldots, c_n)$ to belong to I, else $p(c_1, c_2, \ldots, c_n)$ does not belong to I. All these ground instances of atoms appearing in D determine a set I of ground atoms. Now, I is also a subset of the Herbrand base of D. I is even an Herbrand interpretation which makes D true because of the correspondence of truth in I and in M. So, I is a

Herbrand model of D. A detailed proof can be found in [Swa94] (proof of Theorem 26.5).

The previous remarks show that when a set of clauses has a model, that set has also a Herbrand model. Together with the definition of satisfiability, this result leads to the following proposition (see [Llo87]).

PROPOSITION 1 *Let S be a set of clauses. Then S is satisfiable iff S has a Herbrand model.*

For more general formulas the proposition does not hold (see remarks after proposition 26.5 in [Swa94]). However, when searching for models of deductive databases consisting of clauses only, the proposition states that it is sufficient to observe only Herbrand models.

1.1.4 Query Answering in Databases

A query is an expression representing the information of a database one wants to retrieve. The next two definitions give exactly what information is meant by the query. Such information will be called an answer to the query.

DEFINITION 1.52 Let Q be a query and $X_1, X_2, \ldots, X_n$ the free variables in Q. A *possible answer* to Q is a substitution σ for Q.

DEFINITION 1.53 Let D be a database and M be a model of D. Then a *(correct) answer* to a query Q with respect to D is a possible answer which is true in M.

REMARK Answers to queries do not have to be necessarily ground. But in this thesis an answer to a query is supposed to be ground. So, we are not interested in partially answering questions.

When the query contains symbols which also appear in the database, the query is called *applicable*. When a query is not applicable, the query is meaningless. We suppose that queries are applicable. Applicability of queries does not guarantee the right answer or even an answer at all. For instance, not only we are interested in what information the database provides but also what information the database does not provide. Also, it is possible that in order to answer applicable queries a search through an infinite or undefined part of a relation is needed.

1.1.4.1 Assumptions for Query Answering

Classical negation cannot be implemented here, because it is not feasible to store all negative information of the world in the database. Therefore, negative information is stored implicitly in the database. The database only contains positive information. When queries are posed to such a database some assumptions are made in answering that query, which are important for inferring the implicit negative information of the database. The following three assumptions are made concerning the information content of the database:

- Closed World Assumption (CWA),
- Unique Name Assumption (UNA),

➢ Domain Closure Assumption (DCA).

The CWA states that facts which cannot be found in the database are supposed to be false. The UNA states that individuals with different names in the database are supposed to represent different things in the real world. The DCA states that all the individuals in the database are the only possible ones to choose from.

EXAMPLE 1.6 Consider a relational database consisting of the facts:

$$p(c),\ p(d), m(c), s(j)$$

and the query $Q(X) = \neg(m(X) \wedge p(X))$ expressing the real world situation that *Chris* (c) and *Donald* (d) are parents (p), *Chris* is married (m) and *John* (j) is single (s). The query is posed in order to find persons who are not married parents. The DCA says that in order to answer the query c, d and j are the only individuals to consider. It is obvious that c is not an answer to the query. How about d or j? The UNA says that $\neg(d = c)$ and $\neg(j = c)$ which implies that $m(d)$, $p(j)$ and $m(j)$ cannot be concluded from the database. All facts that cannot be concluded from the database are supposed to be false following the CWA. So, $\neg m(d)$, $\neg p(j)$ and $\neg m(j)$ are true. This implies that the answer to our query is $Q(d)$ and $Q(j)$, i. e. , *Donald* and *John* are not married parents.

Usually, instead of single-character constant and predicate symbols some meaningful constant strings and predicate strings are used in order to show the meaning they should have in the real world. For instance, in the example above we could have used *parent* instead of p.

1.1.4.2 Domain Independent Query Answering

When answering a query long response times or infinite answers are not appreciated. Suppose a database D, consisting of the rule $R : q(X, Y) \longleftarrow r(X), \neg p(Y)$ and some facts with respect to predicates p, q and r. For a query $q(c, Y)$, answers depend on instances of $\neg p(Y)$. Besides the substitutions of Y corresponding to the facts in the database with respect to p all other substitutions lead to an answer. When Y ranges over a whole domain the number of answers is equal to the number of elements in the domain.

Suppose the query was $q(X, c)$, then the problem disappears because by applying the rule Y is bound to c and therefore $\neg p(c)$ does no longer range over the domain as a whole. For the other part of the query, i. e. , $r(X)$, X ranges over the finite set of instances of r. Suppose a variable in a negative literal A of a rule R does not occur in a positive literal of the body of R. The instances of the head of the rule will depend on the database domain as a whole. Application of such rules would require a complete domain search, i. e. , *domain dependent* search, which is in most cases extremely inefficient. In order to keep the evaluation of a rule domain independent, the rules must obey some syntactic property. This syntactical decidable property does not cover the whole class of domain independent formulas, because this class is undecidable. Here, in order to compel the domain independency the subclass of range-restricted formulas (see [Nic82]) suffices for the purpose of this thesis.

DEFINITION 1.54 A formula is *range-restricted* iff each variable that occurs in the head of the formula or in a negative literal of the body of the formula occurs in at least one positive literal of the body of the formula.

Note that this restriction concerns the rules. As a consequence, the queries will behave domain independent as well. Less strong properties are available but are not elaborated here. We could have chosen a less restrictive but more complex property to achieve domain independence, but it does not contribute to the essence of this thesis. The most important result is that the evaluation of queries and rules proceeds independently from the domains. Therefore, rules of a database are supposed to be range-restricted. This will instantiate a negative literal completely whenever the positive literals in the body are instantiated. Other syntactic properties on formulas to guarantee domain independence are available, such as *allowed formulas*, *range separable formulas*, *safe formulas*, *evaluable formulas*, *generalized range-restricted formulas* (see [Cod72,Dec87,Dem82, Top87, Ull88a]).

1.2 Logic and Deductive Databases

From a logical point of view, a relational or deductive database is looked at in two different ways (see [GMN84]). In the first place, there is the *model-theoretic view*. Here, the facts (resp. all facts and deducible facts) of the relational database (resp. deductive database) are looked at as an interpretation (or model) of the set of its logical formulas. Here, the interpretation assigns truth values to these logical formulas. If they are true in this interpretation then the database is a *model* for the formulas. In the second place, there is the *proof-theoretic view* in which the database is looked at as a first-order theory.

In the theoretical part of this paper we use the model-theoretical view. In the practical part of the thesis the proof-theoretical view is used.

1.2.1 Model Theory and Deductive Databases

In determining the semantics of databases, *minimal* Herbrand models (also called *least* Herbrand models, [EK76]) play an important role (see [EK76]). Mimimal Herbrand models of a set of formulas S are smallest Herbrand models in the sense that they contain no other Herbrand models. In general, minimal Herbrand models do not express the intended meaning of deductive databases, as we will see below. Sometimes some other semantics are needed to express the intended meaning of more complex deductive databases. For instance, an extension of the minimal Herbrand model semantics, the perfect model semantics (see [PP90, Prz87, Prz88]).

1.2.1.1 Model Theory for Definite Deductive Databases

In this section only model theory for definite deductive databases is considered. In general, definite deductive databases may have several models. But not all of these models represent the intended meaning of the database. In order to find the intended meaning of a database one tries

to find only minimal Herbrand models. Because of the definite nature of the database only one minimal Herbrand model is found. The next example gives insight in the model theory of definite deductive databases.

EXAMPLE 1.7 Suppose a definite deductive database consists of the following fact and rule base:

$$employed(fred)$$

$$married(fred, wilma)$$

$$dependent(X, Y) \longleftarrow married(Y, X), employed(Y)$$

For this database several models exist. For instance, a model where both $fred$ and $wilma$ are married to each other and are employed and dependent on each other. This is the largest possible Herbrand model for this example. However, this model does not express the intended meaning of the database. The model that expresses the true intension of the database, is the model in which $fred$ married $wilma$, $fred$ is employed and $wilma$ depends on $fred$. As a matter of fact this is also the only minimal Herbrand model and the only minimal one for this database.

In general, in definite deductive databases the (only) minimal model describes the semantics of the database adequately (see theorem 26.6 and 26.8 in [Swa94] and [Vol87]).

1.2.1.2 Model Theory for Indefinite Deductive Databases

In this section only model theory for indefinite deductive databases is considered. As a simplification the indefinite deductive database is supposed to be positive. When negation is involved the determination of models becomes more complicated. This complication in the case of negation is postponed to 1.2.1.3. Indefinite deductive databases have some typical problems related to the indefinite nature of the database, which is clarified at best by using positive indefinite deductive databases only. Because of the indefinite nature of the database several minimal Herbrand models could be found. This means that indefinite deductive databases could have several intended meanings. The next example gives insight in the model theory of indefinite deductive databases.

EXAMPLE 1.8 Suppose an indefinite deductive database consists of the following fact and rule base:

$$employed(fred)$$

$$married(fred, wilma)$$

$$dependent(X, Y) \vee employed(X) \longleftarrow married(Y, X), employed(Y)$$

In this example a person is called dependent or employed if the person's partner is employed. As in the previous example, the model which states all possibilities, i. e. , both $fred$ and $wilma$ are married to the other and are employed and dependent on the other, is not the intended meaning of this database. However, there is not one intended meaning in this case, because $wilma$ is dependent of $fred$ or is employed. In this case, two minimal models are appropriate as a meaning to this database. One in which $fred$ is married to $wilma$, $fred$ is employed and $wilma$ is dependent of $fred$. And one in which $fred$ is married to $wilma$ and both $wilma$ and $fred$ are employed.

In general, each positive indefinite deductive database has one or more minimal Herbrand models, which describe the semantics of the database adequately. For indefinite deductive databases with negation, the minimal models fail to describe the intended meaning of the database as the case of normal deductive databases shows. More on the model theory for indefinite deductive databases can be found in [Prz90], [Prz91] and [Vol89].

1.2.1.3 Model Theory for Normal Deductive Databases

As we have seen, definite deductive databases have a well defined meaning. In this section, we study the intended meaning of a database when negation is introduced. The next example gives insight in the model theory of normal deductive databases with an unrestricted use of negation.

EXAMPLE 1.9 Suppose a normal deductive database consists of the following fact and rule base:

$$\mathit{employed}(\mathit{fred})$$

$$\mathit{married}(\mathit{fred}, \mathit{wilma})$$

$$\mathit{dependent}(X, Y) \longleftarrow \neg \mathit{employed}(X), \mathit{married}(Y, X), \mathit{employed}(Y)$$

In this example a person is called dependent if the person is not employed and that person's partner is employed. As in the indefinite case two minimal models are found. In fact, these are the same models as in the indefinite case. However, the intended meaning differs from the previous case because of the negated presence of $\mathit{employed}(X)$ in the body of the rule. In the first minimal model wilma is dependent of fred and in the latter she is employed. Because in databases the $\neg$ is interpreted as "is not part of the database", the database implicitly contains the information $\neg\mathit{employed}(\mathit{wilma})$ because $\mathit{employed}(\mathit{wilma})$ is not part of the database. Therefore, the intended meaning of this database corresponds to the first minimal model.

REMARK Suppose the rule in the example was replaced by:

$$\mathit{dependent}(X, Y) \longleftarrow \mathit{not\_employed}(X), \mathit{married}(Y, X), \mathit{employed}(Y)$$

where *not_employed* is a predicate which explicitly states that a person is not employed. Using this predicate makes it is possible to store all such facts in the database. Then the normal deductive database has turned into a definite one for which the only minimal model is the model expressing that fred is married to wilma and fred is employed. By formulating a negative relation in a positive way, positive and negative atoms play a symmetric role in the database. Note that this corresponds to a more logic-like interpretation of $\neg$ in databases for which minimal Herbrand models describe the intended meaning of such a database well. However, negation is still needed to express that certain information is not available in the database. This kind of use of negation is explored later.

REMARK While equivalent formulas in first-order logic have the same meaning in logic, in a deductive database setting equivalent formulas may imply a completely different meaning for the database. As we saw from EXAMPLE 1.8 and EXAMPLE 1.9 the rules that were used in the examples are equivalent in the logical sense but the intended meaning of both databases differs. The

appearence of an atom as a negated atom in a conjunction of the body of a rule has a greater discriminating effect on the set of minimal models than the corresponding occurrence in the disjunction of the head. To be more precise, the set of minimal models of a database of the first kind is a subset of the set of minimal models of the corresponding database of the second kind. (see [Prz87]).

Minimal Herbrand models are a good way to describe the semantics of a positive database; however, it turned out that in normal databases the minimal Herbrand models do not necessarily correspond to the intended meaning of these normal deductive databases. So, some semantics should be developed to take the semantics of normal deductive databases into account as well. This has been done for a subclass of the class of normal deductive databases, the stratified normal deductive databases (see [ABW88]), in which the use of negation is restricted.

1.2.1.4 Model Theory for Stratified Deductive Databases

When introducing negation in deductive databases problems arise. In the following only normal deductive databases are considered. In order to describe the problems involving negation, first fixpoints of mappings on a partial ordered set are introduced (see [Llo87]). In the previous examples the models of the databases in the examples were given. But how can we find those models. Fixpoint theory is very helpful here. It turns out that the fixpoints of certain mappings related to the database are equal to the minimal models of that database.

DEFINITION 1.55 A *partially ordered* set S is a set on which a relation $r : S \times S \rightarrow \{true, false\}$ is defined that satisfies the following conditions:

(i) for all $x \in S : r(x, x)$ is true,

(ii) for all $x, y \in S$: if both $r(x, y)$ and $r(y, x)$ hold, i. e. , are true, then $x = y$,

(iii) for all $x, y, z \in S$: if both $r(x, y)$ and $r(y, z)$ hold, i. e. , are true, then $r(x, z)$ is true. Relation r is called a *partial order* (on set S).

Here, the partially ordered sets of interest are the powersets 2^S for some set S on which the subset relation ($\subseteq$) is defined.

DEFINITION 1.56 Let r be a partial order on set S and $X \subseteq S$. Then $u \in S$ is an *upper bound* of X if $r(x, u)$ holds for all $x \in X$. When for the upper bound u of X it also holds that $r(u, u')$ for all upper bounds u' of X, u is called the *least upper bound* of X. Similarly, $l \in S$ is a *lower bound* of X if $r(l, x)$ holds for all $x \in X$. When for the lower bound l of X it also holds that $r(l', l)$ for all lower bounds l' of X, l is called the *greatest lower bound* of X.

DEFINITION 1.57 S is called a *complete lattice* with respect to a partial order on S if for every subset X of S a least upper bound and greatest lower bound exist.

Note that for a set S, the partial order $\subseteq$ on 2^S gives a complete lattice because the least upper bound of a collection of subsets of S can be found by taking the union over this collection, while the greatest lower bound can be found by taking the intersection over this collection.

DEFINITION 1.58 Let S be a complete lattice with respect to partial order r. A mapping $T : S \rightarrow S$ is *monotonic* if for all $x, y \in S$ for which $r(x, y)$ holds also $r(T(x), T(y))$ holds.

DEFINITION 1.59 Let $T : S \rightarrow S$ for a set S. A *fixpoint* of T is an element s of S for which $T(s) = s$.

DEFINITION 1.60 Let S be a complete lattice with respect to partial order r. Suppose X is the subset of S consisting of all fixpoints of a mapping $T : S \rightarrow S$. Now, an element s of S is called a *least fixpoint* (resp. a *greatest fixpoint*) if it is a lower bound (resp. upper bound) of X.

REMARK Tarski (see [Tar88]) showed that if a mapping T of a complete lattice is monotonic then T has a least fixpoint and a greatest fixpoint. Our interest goes to the least fixpoint of such a mapping.

The theory of fixpoints is applied to the theory of databases as follows. Let D be a deductive database. Take as initial set the Herbrand base H of D with as partial order on the powerset of H the set inclusion $\subseteq$. Now 2^H is a complete lattice with respect to $\subseteq$. The interesting mapping from 2^H to 2^H is the mapping, say T_D, which assigns to each element $I \in 2^H$, i. e. , $I \subseteq H$, a subset which is the union of I and the set, say $T_D(I)$, of all derivable facts by applying each rule of the database once to the facts in the subset. To be more precise, $T_D(I) = \{A\theta | A \longleftarrow L_1, L_2, \ldots, L_n \in D$ and θ is a ground substitution of $L_1, L_2, \ldots, L_n$ such that $L_i\theta \in I$ for each $i = 1, 2, \ldots, n\}$. Lloyd defined a more general mapping for all kinds of interpretations and for a broader class of databases (see [Llo87]). Here, we are only interested in a mapping based on Herbrand interpretations and function free databases consisting of normal clauses. In case of a deductive database there is a close relationship between Herbrand models and fixpoints. Namely, for each $I \subseteq H$ it holds that I is a Herbrand model of D iff I is a fixpoint of T_D iff $T_D(I) \subseteq I$ (see [Llo87]). The mapping T_D is monotonic. So, from the previous remark it follows that this database D guarantees that a Herbrand model I of D is a fixpoint of T_D. The minimal Herbrand models of a deductive database D correspond to the least fixpoints of T_D.

Databases with negation may have several minimal fixpoints (minimal Herbrand models) (see EXAMPLE 1.9). A database is only useful when the database has only one unique fixpoint (minimal Herbrand model). In order to compel a unique fixpoint (minimal Herbrand model) the database is supposed to obey the syntactic property of stratification.

DEFINITION 1.61 A database is *stratified* if we can partition the database in $P_1, P_2, \ldots, P_n$, such that for each $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, n$:

(i) if A is an atom occurring in the body of a clause in P_i, then the definition of the predicate symbol of A can only appear in P_j for some $j \leq i$,

(ii) if $\neg A$ is a negated atom occurring in the body of a clause in P_i then the definition of the predicate symbol of A can only appear in P_j for some $j < i$,

In case of a stratified deductive database there is a close relationship between Herbrand models and fixpoints. Namely, for each $I \subseteq H$ it holds that I is a Herbrand model of D iff I is a fixpoint

of T_D iff $T_D(I) \subseteq I$ (see [Llo87]). In fact, stratification makes the mapping T_D monotonic. So, from the previous remark it follows that stratification of a database D guarantees that a Herbrand model I of D is a fixpoint of T_D.

So, one has succeeded in finding a semantics for a subclass of the class of normal deductive databases, the stratified deductive databases (see [ABW88]) and *locally stratified deductive databases* (see [Prz87]), for which the perfect models (see [Prz88]) give a uniquely determined semantics for the database. Other semantics for subclasses of normal deductive databases are the *weakly perfect model semantics* (see [PP90]), the *stable model semantics* (see [GL88]) and the *well-founded model semantics* (see [GRS88]). Przymusinski (see [Prz90]) extended both the perfect model semantics and the well-founded model semantics to the *stationary semantics* applicable to the class of all indefinite deductive databases and normal deductive databases. Other semantics for indefinite deductive databases are the *extended well-founded semantics* (see [Ros90b]), the *generalized well-founded semantics* (see [BLM90]) and the *disjunctive* and *partial disjunctive stable semantics* (see [Prz90]). In this thesis we consider stratified deductive databases, which does not mean that the results presented in this thesis cannot be extended to more general kinds of databases.

1.2.2 Proof Theory and Deductive Databases

In contrast to the model-theoretic approach in which the meaning of a database is given, the proof-theoretic approach tries to infer syntactically information from a database.

The model-theoretic approach has some disadvantages when describing more advanced databases. For instance, databases with incomplete information, for example indefinite deductive databases or databases with *null values* representing values which are not known in the present database state, get a temporary value. Null values represent any value of a domain. So, one cannot say for sure if the null value is equal to a certain value in the domain or not. Because a model shows only exactly one database state, one model is not sufficient to represent several possible database states corresponding to one fixed value for each null value. So, the model theory will fail here. In the case of indefinite deductive databases it is often not possible to describe the database model-theoretically by only one set of ground literals, i. e. , several models may be needed. In the proof-theoretic approach one theory is enough to cover several models describing one database state. The proof-theoretic approach turns out to be a more natural approach to deal with various extensions of the relational model.

The proof-theoretic approach takes first-order logic as a basis which consists of a first-order language, a set of *logical axioms*, i. e. , formulas that represent some basic valid statements in logic from which other valid statements can be built, and some *inference rules*. By adding to the logical axioms some first-order formulas, i. e. , the *non-logical axioms*, the first-order predicate calculus becomes a *first-order theory*. When these first-order formulas represent the database, the first-order theory is also called a *database theory*.

DEFINITION 1.62 Let $\mathcal{T}$ be a first-order theory. Let S be a set of formulas. When F is the result of finitely many applications of the inference rules in $\mathcal{T}$ on the axioms in $\mathcal{T}$ and formulas in S, then F is called *derivable* from S in the first-order theory $\mathcal{T}$, notation $S \vdash_{\mathcal{T}} F$. When S is empty, F is called a *theorem* of $\mathcal{T}$, $\vdash_{\mathcal{T}} F$. A *proof* that F is derivable from S is a sequence of intermediate formulas derived by the successive application of the inference rules untill F is reached.

When it is clear which theory $\mathcal{T}$ is meant the subscript $\mathcal{T}$ is deleted.

When one wants to rely on the proof-theoretic approach one has to be sure that the inference corresponds to the intended meaning from the model-theoretic point of view. The set of inference rules should obey the following properties.

DEFINITION 1.63 An inference system of a first-order theory $\mathcal{T}$ is called *sound* iff for any set of formulas S and any formula F, if $S \vdash F$, then $S \models F$.

DEFINITION 1.64 An inference system of a first-order theory $\mathcal{T}$ is called *complete* iff for any set of formulas S and any formula F, if $S \models F$, then $S \vdash F$.

Because in this thesis we are mainly interested in databases consisting of clauses, an inference rule is used which is often used for clauses, namely the resolution inference rule, see [Rob65]. Other theorem proving techniques are available, for instance the tableaux based proof procedure (see [Oph92, SO90, SO93]). The resolution inference rule is based on finding a proof by refutation, i. e. , in order to show that $S \vdash F$ one shows that S together with $\neg F$ are not satisfiable in the theory $\mathcal{T}$ at the same time. Assuming that both are satisfiable and the knowing that a resolution step transforms two satisfiable clauses into another satisfiable clause, one tries to infer the empty clause. In that case, a contradiction, i. e. , the empty clause, is inferred. Therefore, inferring the empty clause from S and $\neg F$ corresponds to a proof that from S formula F can be derived. This kind of proof is called a *refutation*. The negated formula that has to be refuted is called the *goal*. The refutation ends in an empty clause, which is a clause that is unsatisfiable.

This inference rule is known to be sound and complete. However, when the clause is not a theorem the search for a proof may never end.

When concentrating on deductive databases, from a proof-theoretic point of view a theory is constructed which represents the database in such a way that each formula derivable from the theory is true from a model-theoretic view of the database.

The non-logical axioms, which represent the deductive database, consist of the

- *particularization axioms*; consisting of axiomizations of
  - the domain closure assumption,
  - the unique name assumption,
  - the closed world assumption,

- the equality predicate used in axioms describing the previous assumptions,

➢ axioms representing the facts of the database,

➢ axioms representing the rules of the database.

EXAMPLE 1.10 Consider a database D consisting of two predicate symbols, p and q, and n constant symbols, $c_1, c_2, \ldots, c_n$. Suppose D consists of the facts:

$$p(c_1), \ldots, p(c_{n-1}), q(c_n)$$

and the rule:

$$\forall X[p(X) \rightarrow q(X)].$$

The formula $\forall X q(X)$, which is true in this database, cannot be derived from the database until it is made clear where $\forall X$ stands for in this database. Therefore an expression

$$\forall X[X = c_1 \vee X = c_2 \vee \cdots \vee X = c_n]$$

is added to the theory representing the database. So, the database will automatically obey the domain closure assumption. However, it is still impossible to derive from the theory that $\neg(c_i = c_j)$ for all $i, j \in \{1, 2, \ldots, n\}$ with $i \neq j$, while this should be the case because of the unique name assumption. Therefore for each $i, j \in \{1, 2, \ldots, n\}$ and $i \neq j$ an inequality $\neg(c_i = c_j)$ is explicitly added to the theory.

Further, in order to be able to derive $\neg p(c_n)$ (interpreted as $p(c_n)$ cannot be derived) from the theory established so far, first it must be made explicit for which X $p(X)$ can be derived. This formalisation of the closed world assumption is established by the addition of first order formulas for each predicate symbol, called the *completion axioms* for that predicate. The completion axiom for p is:

$$\forall X[p(X) \rightarrow X = c_1 \vee X = c_2 \vee \cdots \vee X = c_{n-1}]$$

and for q:

$$\forall X[q(X) \rightarrow p(X) \vee X = c_n].$$

Note that it is possible to reformulate facts in the database as an implication. For instance, $p(c_1)$ can be rewritten as $\forall X[X = c_1 \rightarrow p(X)]$. Or when representing all facts with predicate symbol p together in one formula we get:

$$\forall X[X = c_1 \vee X = c_2 \vee \cdots \vee X = c_{n-1} \rightarrow p(X)].$$

Therefore, in general all facts with respect to a predicate symbol together with its completion are often written as equivalences. So, in our example for predicate symbol p the following expression replaces the completion axiom and the facts with respect to p:

$$\forall X[p(X) \leftrightarrow X = c_1 \vee X = c_2 \vee \cdots \vee X = c_{n-1}]$$

Now, it is possible to derive $\neg p(c_n)$ by using the completion rules.

Here, an equality predicate symbol was introduced without any explicit knowledge of its use. So, from these formulas it is not possible to prove that for c_1 the expression $X = c_1 \vee X = c_2 \vee \cdots \vee X = c_n$ can be proved. Therefore the instances with respect to the equality predicate are also made explicit, i. e. , $c_1 = c_1, c_2 = c_2, \ldots , c_n = c_n$ are added to the theory. Further, several other well known equality laws, like *symmetry*, *transitivity* and *Leibniz' substitution principle of equal terms*, are axiomitized and added to the theory in order to be able to use equality in conformity with its meaning. Note that an infix notation is used for this special kind of predicate instead of a prefix notation in case of ordinary predicates.

Because of the explosive growth of the number of particularization axioms when the number of facts and relations in the database increases, one has searched for a practical solution. In order to avoid the axioms that comprise the closed world assumption and the unique name assumption, the meta-rule *negation as finite failure* is used (see [Cla78]). This rule states that when a formula cannot be proved, it is possible to infer the negated formula from the theory. Further it is possible to avoid the axioms concerning the domain closure assumption by making the database domain independent. This is done here by demanding that the formulas representing the database must obey the syntactic property of range-restrictiveness. Because axioms concerning the equality predicate were only needed for the other particularization axioms, these axioms can also be avoided. So, our proof-theory now consists of:

- ➢ axioms representing the facts of the database,
- ➢ range-restricted axioms representing the rules of the database,
- ➢ a resolution inference rule together with a negation as finite failure meta-rule.

In order to establish an operational proof-system, the SLDNF-resolution rule has proved to be a satisfactory inference rule, which is a basis for many logic programming languages (see [Llo87, Swa94]).

The logic programming language Prolog is based on an implementation of the SLDNF proof procedure. It is possible to look at pure Prolog as SLDNF with the following order constraints:

- ➢ the order of the clauses in the logic program is the order in which the clauses are chosen in order to resolve a goal with a literal of the chosen clause,
- ➢ the left most matching literal is chosen in order to get the next resolvent.

So, it is possible to implement a deductive database fully in Prolog, because all axioms and the inference rule can be described by Prolog. In this thesis, Prolog is powerful enough to implement small deductive databases. The next section shows that Prolog is not powerful enough to implement an efficient deductive database management system; even when Prolog cooperates with a relational database management system, see [CGW89] and [SW86], it is not sufficient.

1.3 Logic + Database System < Deductive Database System

Nowadays, fact bases of expert systems, rule based systems and logical programming languages are becoming often too big to be loaded into main memory. So, there is a growing need to keep

databases on secondary storage. Also, several efforts for making relational databases more powerful, for instance, by incorporating rules in the datamodel, were made. In order to do so, relational database management systems were interfaced to expert systems, rule based systems and logic programming languages (see [CGT90]). In most cases, the communication between the database and the system's programming language proceeds via the intermediate language SQL (Structured Query Language). Data manipulation statements are converted automatically to SQL statements which are sent to the relational database management system and handled there. The result is returned to the systems programming language. However, interfacing a logic to a database does not establish a deductive database, while implementing a deductive database management system is far from trivial. It turns out that logic is a very natural tool to describe databases, but that it is certainly not a natural one to answer queries efficiently. Before expanding on this, first two kinds of interfaces of a programming language and a relational database are distinguished.

1.3.1 Coupling Logic to Database Systems

In order to get a logical database one could interface a logical programming language to a relational database. The idea is to take advantage of the functionalities of both the programming language and the relational database, which guarantees the security, concurrent handling and recoverability of data, in order to create a powerful combined system. Inferencing is done at the programming level while query answering is done at the database level. In general, two kinds of systems, consisting of an existing programming language and an existing database system connected by an interface, are distinguished, namely *loosely coupled systems* and *tightly coupled systems*.

1.3.1.1 Loosely Coupled Systems

A system consisting of a programming language interfacing a database is called a *loosely coupled system* when the programming language, in order to obtain facts from the database, first copies the stored structures containing those facts from the database in the working memory of the programming language. Herein, the needed facts are selected from those structures. For instance, in case of a relational database and a logic programming language, when one tries to find an instance of a database relation, the whole relation is first transfered to the memory of the logic programming language. Hereafter, the specific fact is selected without using the relational database management system. So, the query is answered within the programming language.

It is possible to see Prolog as a programming language separated from its own fact base. When looking at Prolog this way, i. e. , as a programming language for which the whole fact base is (already) loaded in the Prolog memory, Prolog can be looked at as loosely coupled to its own fact base.

It is obvious that the loosely coupled systems may have severe disadvantages in case of very large databases. Database operations involving large parts of the database (for instance, joins) have to

be executed in the programming language memory lacking all the efficiencies a database management system offers.

1.3.1.2 Tightly Coupled Systems

A system consisting of a programming language interfacing a database is called a *tightly coupled system* when the programming language uses the facts in the database directly, i. e. , during the inference process of the programming language. So, facts are obtained from the database just when it is needed in the inference process. This means from the database's point of view that it looks as if the programming language is an ordinary user. It also means that each time the program needs a fact or wants to store a fact, the database has to be accessed.

REMARK The programming language Prolog can use its own database because the prolog database is completely available in main memory. When looking at Prolog as a programming language connected to its own Prolog database, where database operations have a direct effect on the database, Prolog can also be seen as tightly coupled to itself.

Operations performed by the programming language on data of the database correspond directly to available operations in the database management system. For instance, in case of a relational database and a logic programming language, when one tries to find an instance of a database relation, the instance is found by first translating the query in the programming language to a corresponding query, often in SQL, in the database management system. So, here the query is answered by the database management system.

The tight coupling enables the user to get all the functionalities a database management system offers, such as efficient data manipulation. However, the tight coupling of a logical programming language and a relational database does not guarantee efficient query evaluation. For instance, when Prolog is tightly coupled to a relational database management system, for which each query is stated in Prolog and the subgoals of the query are evaluated in a Prolog-like manner (i. e. , the subgoals in the query are evaluated in order of appearance) each subquery is handled efficiently but the overall query is rather cumbersome. The next example will illustrate this.

EXAMPLE 1.11 Suppose D is a relational database with a *father* and a *student* relation, where numbers represent people, consisting of the following facts:

$$father(1, 10),\ father(1, 11),\ father(2, 20),\ father(3, 30),\ father(3, 31),\ father(4, 40),$$

$$student(2),\ student(3),\ student(10),\ student(30),\ student(31)$$

For instance, fact $father(1, 10)$ represents that father 1 has child 10.

Consider the query ?- $student(X)$, $father(1, X)$ which represents the query "Is there a student who has 1 as father?". When posing this query the subquery $student(X)$ is answered first. It is possible to answer this subquery by the whole relation *student*, in this case five students. For each answer of

the first subquery, the second subquery is answered. For example, if *student*(2) is found as an answer to the first subquery the second subquery reduces to *father*(1, 2). So, the database is accessed in order to look for a fact *father*(1, 2). The fact base is searched until the answer *student*(10) leads to a positive answer to the overall query. Consider the query ?- *father*(1, X), *student*(X) which is semantically equivalent to the previous query. Procedurally however, this query will lead to the right answer in only two database accesses. Intuitively, this procedural difference is clear because, knowing for which father a student must be found, the set of possible students is reduced to the children of 1 instead of all students.

REMARK The order in which the facts are stored can make a difference in the number of database accesses. In the case of the first query with a little luck the *student*(10) fact was the first one to access. In that case, only two database accesses were needed to answer our query. In the worst case, the *student*(10) fact is accessed at last and ten database accesses are needed before the answer to the overall query is reached. In case of the second query the worst case would be the case that *father*(1, 11) is accessed before *father*(1, 10), which results in three database accesses. Note that when the query is universally quantified, i. e. , "find all children of father 1 who are also student", the search continues after a first positive instance of the query has been found. In this case, the number of database facts always equals the number of database accesses.

The subquery order also influences the number of database accesses. In the worst case of the corresponding existentially quantified query, ten accesses in the first case and three accesses in the second case are needed.

So, the number of database accesses may be enormous in the case of tight coupling, while the lack of efficiency is a drawback of loosely coupled systems. Therefore, the two approaches need to be combined in order to get the best of both worlds.

1.3.2 Deductive Database Systems in Practice

When advancing towards deductive database management systems the user should only be concerned with the logic of his program. The control of the program must be the system's responsibility. In the case of coupled systems, the coupling itself is either a responsibility of the user or of the system. When the interface automatically couples the programming language and the database system without any interference of the user, the interface is called *transparent* or in other words, the data are completely transparent to the programming language. The interface is not transparent if the user is responsible for this translation. Note that between those extremes, completely transparent and totally not transparent, several other configurations are possible. For more about architectures and examples of systems that couple Prolog to Relational Systems the reader is referred to Part I of [CGT90], in which these subjects are studied thoroughly. In the appendix a tightly coupled system is presented consisting of Prolog and a relational database system with a (partial) transparent interface in order to simulate a deductive database.

1.3.2.1 Some Deductive Database Systems

Besides Prolog with an interface to a database management system, other (experimental) languages are also available. Special purpose languages are available such as Datalog (see [Ull88a, Ull88b]), $\mathcal{LDL}$ ($\mathcal{L}$ogical $\mathcal{D}$ata $\mathcal{L}$anguage, see [TZ86]), NAIL! (**N**ot **A**nother **I**mplementation of **L**ogic**!**, see [DMP93,MGU86]), and in the programming languages offered by the deductive database management systems XSB (see [SSW94]) and CORAL (see [RRS+93, RSSS93]). These languages are designed as declarative database application languages *and* query languages. They are syntactically comparable to Prolog, but do not have the order problems and special control predicates to influence the inference as in Prolog. The control in these languages is the responsibility of the underlying deductive system.

Datalog does not contain any function symbols. In the early days of Datalog it was a language without negation. Some people expanded the expressiveness of the language by adding negation (see [AH88]). When comparing Datalog and Prolog at an inference level, we see that Prolog uses a depth first search strategy while Datalog, in most cases, uses a breadth first search strategy in order to produce set-oriented answers to queries.

The logical database language $\mathcal{LDL}$ developed at MCC (Microelectronics and Computer Technology Corporation, Austin, Texas), is also an alternative for implementing a deductive database. This language, proposed by Tsur and Zaniolo in [TZ86], allows function symbols. Also some computational shortcomings of Prolog as a database query language are handled by $\mathcal{LDL}$ (see [NT89, ST91, Tsu88]). Further, the NAIL! system (see [MGU86, Ull88b]) is a knowledge-based system, developed at Stanford University, which also offers the possibility of expressing queries in a logical language. The semantics of deductive databases in $\mathcal{LDL}$ and NAIL! are based on perfect models (see [Prz88]).

People familiar with Datalog, $\mathcal{LDL}$ and/or the NAIL! system could therefore read the Prolog programs in this thesis as Datalog, $\mathcal{LDL}$ and/or NAIL! programs. As we have seen earlier, the coupling of a programming language to a database language is not always the best solution. Control is often a part of the programming language and therefore the user's responsibility. In case of integration of logic into databases, one searches for a real deductive database management system that incorporates the logic programming language into the system, such that the system becomes responsible for an efficient evaluation of logic queries. Therefore, a Deductive Database Management System should be more than a logic programming language coupled to a Database Management System. This means that some query optimization techniques for queries must be integrated into the deductive database management system. For instance, EXAMPLE 1.11 showed that it might be better to evaluate an instantiated subquery before an uninstantiated one. Some of these techniques are mentioned in the next section.

1.3.2.2 Query Optimization in Deductive Database Systems

Logic is also a powerful language to express queries and computations in the relational datamodel. Its expressiveness goes beyond that of SQL when (recursive) rules are involved.

Several optimizations (see [SS86]) are possible to reduce the time to answer a query. As stated before, we can read Prolog programs as Datalog, $\mathcal{LDL}$ or NAIL! programs. Systems working with Datalog or $\mathcal{LDL}$ and the NAIL! system take the responsibility for the evaluation of a query. Their query evaluator applies some general optimization techniques in order to answer queries more efficient. In Prolog the order of subgoals of a query determines the order of the evaluation of the subqueries. So, the user, who writes the query down, is responsible for the way a query is evaluated. However, some of the optimization techniques of Datalog (see [AH88,RBK88]), $\mathcal{LDL}$ (see [NT89, Tsu88, TZ86]) and NAIL! (see [MGU86, Mor88]) programs can easily be incorporated into Prolog programs. This means that the programmer must add some meta-level control in order to control the natural evaluation of a query. For instance, it is a better choice to put a subgoal in front if it may lead to an early failure of the subgoal. Here, the following considerations are important with respect to early failure of a subgoal:

- subgoals in the query which are not (partially) instantiated are evaluated after fully instantiated subgoals,
- subgoals involving base predicates are placed before subgoals involving derived predicates,
- subgoals leading to a few possible answers should be placed before subgoals that may lead to a great number of answers,
- negated subgoals will only be evaluated when they are fully instantiated.

In the worst case the first subgoal in a query, say G, has no variables in common with the update literal in the head of a rule. The update leads to a search through the whole relation of G. If there is also a subgoal G' in this clause which is instantiated partially and also has one or more variables in common with G, then subgoal G' must be evaluated before G. By evaluating G' first, G is also instantiated before G is evaluated; this prevents a full search through the extension of the relation.

Several other optimization strategies are:

- naive evaluation, (see [BR86]),
- semi-naive evaluation, (see [BR86]),
- iterative query-subquery, (see [BR86, Vie86]),
- recursive query-subquery, (see [BR86, Vie86]),
- Henschen-Naqvi strategy, (see [BR86, BMSU86, HN84]),
- Prolog-strategy, (see [BR86]),
- APEX strategy, (see [BR86, Loz85, Vie86]),
- the Alexander Method, (see [RLK86]),
- the RQA/FQI strategy, (see [Nej87]),

- Aho-Ullman strategy, (see [BR86]),
- Kifer-Lozinskii (static filtering) strategy, (see [BR86, KL86, Loz85]),
- magic set strategies, (see [BR86, BR87, BMSU86, KL86, MP94, Ros90a, SZ86]),
- counting methods, (see [BR86, BR87, BMSU86, GZ92, HN88, KL86]),
- reverse counting methods, (see [BR86, BMSU86]).

This section showed that Prolog gives answers to queries in a tuple-oriented way while in real deductive databases answers are more set-oriented. Also, in Prolog the order of the facts and rules and special control predicates determine the order and the efficiency of answers to queries, while in deductive databases query answering should be order-independent. So, although Prolog was meant to be declarative, it has some procedural flavour. A deductive database should be purely declarative. This justifies the following paradigm:

Logic + DataBase System < Deductive DataBase System

References

[ABW88] KRYSZTOF R. APT, HOWARD A. BLAIR AND ADRIAN WALKER. Towards a Theory of Declarative Knowledge. In J. MINKER, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148, 1988.

[AH88] S. ABITEBOUL AND R. HULL. Data functions, DATALOG and Negation. In *ACM SIGMOD International Conference on Management of Data*, pages 143–153, 1988.

[BJ93] PETER BUNEMAN AND SUSHIL JAJODIA, editors. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22-2, Washington, DC, June 1993. ACM Press.

[BLM90] C. BARAL, J. LOBO AND J. MINKER. Generalized Well-founded Semantics for Logic Programs. In *Proceedings of the Tenth International Conference on Automated Deduction*, pages 102–116, West-Germany, 1990.

[BMSU86] FRANÇOIS BANCILHON, DAVID MAIER, YESHOSHUA SAGIV AND JEFFREY D. ULLMAN. Magic Sets and other strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGART-SIGMOD Symposium on Principles of Database Systems*, volume I, pages 1–15, March 1986.

[BR86] FRANÇOIS BANCILHON AND RAGHU RAMAKRISHNAN. An Amateur's Introduction to Recursive Query Processing Strategies. In CARLO ZANIOLO, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 15 of *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52. ACM, June 1986.

[BR87] CATRIEL BEERI AND RAGHU RAMAKRISHNAN. On the Power of Magic. In *Proceedings of the Sixth ACM SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 269–283, March 1987.

[CGT90] S. CERI, G. GOTTLOB AND L. TANCA. *Logic Programming and Databases.* Springer-Verlag, 1990.

[CGW89] S. CERI, G. GOTTLOB AND G. WIEDERHOLD. Efficient Databases access through PROLOG. *IEEE Transactions on Software Engineering*, 15(2):153–164, 1989.

[Cla78] K. L. CLARK. Negation as Failure. In H. GALLAIRE AND J. MINKER, editors, *Logic and Databases*, pages 293–322, New York, 1978. Plenum Press.

[Cod70] E.F. CODD. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod72] E. F. CODD. Relational Completeness of Data Base Sublanguages. In R. RUSTIN, editor, *Data Base Systems*, pages 65–98, Englewood Cliffs, NJ, 1972.

[Cod90] E. F. CODD. *The Relational Model for Database Management; version 2.* Addison-Wesley, 1990.

[Dat95] C. J. DATE. *Relational Database, writings 1991-1994.* Addison-Wesley, 1995.

[Dec87] H. DECKER. Integrity Enforcement on Deductive Databases. In LARRY KERSCHBERG, editor, *Expert Database Systems: Proceedings from the First International Conference*, pages 271–285. Charleston, Sc., 1987.

[Dem82] R. DEMOLOMBE. Syntactical Characterization of a Subset of Domain Independent Formulas. Technical report, ONERA-CERT, Toulouse, 1982.

[DMP93] MARCIA A. DERR, SHINICHI MORISHITA AND GEOFFREY PHIPPS. Design and Implementation of the Glue-Nail Database System. In Buneman and Jajodia [BJ93], pages 147–156.

[EK76] M. VAN EMDEN AND R. KOWALSKI. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 4(23), 1976.

[GH86] G. GOOS AND J. HARTMANIS, editors. *International Conference on Database Theory, '86 (ICDT '86)*, volume 243 of *Lecture Notes in Computer Science*, Rome, Italy, 1986. Springer-Verlag.

[GL88] M. GELFOND AND V. LIFSCHITZ. Proceedings of the Fifth Logic Programming Symposium. In R. KOWALSKI AND K. BOWEN, editors, *Minimal Model Semantics for Logic Programming*, pages 1070–1080, Mass., USA, 1988. MIT Press.

[GMN84] H. GALLAIRE, J. MINKER AND J.-M. NICOLAS. Logic and Databases, a Deductive Approach. *Computing Surveys*, 16(1):154–185, 1984.

[GRS88] A. VAN GELDER, K. A. ROSS AND J. S. SCHLIPF. The Well-Founded Semantics for General Logic Programs. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 221–230, March 1988.

[GZ92] S. GRECO AND C. ZANIOLO. Optimization of Linear Logic Programs using Counting Methods. In *Lecture Notes in Computer Science*, volume 580, pages 72–87, 1992.

[HN84] L. HENSCHEN AND S. NAQVI. On Compiling Queries in Recursive First-Order Data Bases. *Journal of ACM*, 31(1):47–85, 1984.

[HN88] R. HADDAD AND J. NAUGHTON. Counting methods for Cyclic Relations. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 333–340, 1988.

[Ker86] LARRY KERSCHBERG, editor. *Expert Database Systems: Proceedings from the First International Workshop*. Charleston, Sc., 1986.

[KL86] MICHAEL KIFER AND ELIEZER L. LOZINSKII. Filtering Data Flow in Deductive Databases. In Goos and Hartmanis [GH86], pages 186–202.

[Llo87] J. W. LLOYD. *Foundations of Logic Programming; Second, Extended Edition*. Springer-Verlag, 1987.

[Loz85] ELIEZER L. LOZINSKII. Evaluating Queries in Deductive Databases by Generating. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 173–177, Los Angeles, California, August 1985.

[Mai83] D. MAIER. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md., 1983.

[MGU86] K. MORRIS, A. VAN GELDER AND J. D. ULLMAN. Design Overview of the NAIL! System. In *Proceedings third International Conference on Logic Programming*, pages 554–568, 1986.

[Mor88] KATHERINE A. MORRIS. An Algorithm for Ordering Subgoals in NAIL! In *Proceedings of the Seventh ACM SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 82–88, March 1988.

[MP94] INDERPAL SINGH MUMICK AND HAMID PIRAHESH. Implementation of Magic-sets in a Relational Database System. In Snodgrass and Winslett [SW94], pages 103–114.

[Nej87] WOLFGANG NEJDL. Recursive Strategies for Answering Recursive Queries – the RQA/FQI strategy. In PETER M. STOCKER AND WILLIAM KENT, editors, *Proceedings of the Thirteenth Conference on Very Large Data Bases*, pages 43–50, Brighton, England, August 1987.

[Nic82] J. M. NICOLAS. Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, 18(3):227–253, 1982.

[NT89] SHAMIM NAQVI AND SHALOM TSUR. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.

[Oph92] W. M. J. OPHELDERS. *Automated Theorem Proving based upon a Tableau-method with Unification under Restrictions: Theory, Implementation and Empirical Results*. PhD thesis, Department of Philosophy, KUB Tilbury, Tilburg, The Netherlands, 1992.

[PP90] HALINA PRZYMUSINSKI AND TEODOR PRZYMUSINSKI. Semantic Issues in Deductive Databases and Logic Programs. In R. BANERJI, editor, *Formal Techniques in Artificial Intelligence*, pages 321–367, North-Holland, Amsterdam, 1990.

[Prz87] TEODOR PRZYMUSINSKI. On the Semantics of Stratified Deductive Databases. In J. MINKER, editor, *Proceedings Workshop on the Foundations of Deductive Databases and Logic Programming*, pages 433–443, Los Altos, 1987. Morgan Kaufmann.

[Prz88] TEODOR PRZYMUSINSKI. Perfect Model Semantics. In *Proceedings of the International Conference on Logic Programming*, pages 1081–1096, 1988.

[Prz90] TEODOR C. PRZYMUSINSKI. Semantics of Disjunctive Logic Programs and deductive Databases. In C. DELOBEL, M. KIFER AND Y. MASUNAGA, editors, *Deductive and Object-Oriented Databases; Proceedings of the Second International Conference, DOOD'91*, volume 566 of *Lecture Notes in Computer Science*, pages 85–107, Munich, Germany, December 1990.

[Prz91] TEODOR C. PRZYMUSINSKI. Stable Semantics for Disjunctive Programs. *New Generation Computing Journal*, 9, October 1991.

[RBK88] RAGHU RAMAKRISHNAN, CATRIAL BEERI AND RAVI KRISHNAMURTHY. Optimizing Existential Datalog Queries. In *Proceedings of the Seventh ACM SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 89–102, March 1988.

[Rei84] RAYMOND REITER. *Towards a Logical Reconstruction of Relational Database Theory*, chapter 8, pages 191–233. Springer-Verlag, 1984.

[RLK86] J. ROHMER, R. LESCOEUR AND J. M. KERISIT. The Alexander Method – a Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing*, 4:273–285, 1986.

[Rob65] J. A. ROBINSON. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[Ros90a] K. A. ROSS. Modular Stratification and Magic Sets for Datalog Programs with Negation. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 161–171, 1990.

[Ros90b] K. A. ROSS. The Well-founded Semantics for Disjunctive Logic Programs. In WON KIM, JEAN-MARIE NICOLAS AND SHOJIRO NISHIO, editors, *Deductive and Object-Oriented Databases*, pages 385–402, Osaka, Japan, 1990. Proceedings of the First International Conference on Deductive and Object-Oriented Databases, DOOD'89, Osaka University, Toyonaka,, North-Holland.

[RRS+93] RAGHU RAMAKRISHNAN, WILLIAM G. ROTH, PRAVEEN SESHADRI, DIVESH SRIVASTAVA, AND S. SUDARSHAN. The CORAL Deductive Database System. In Buneman and Jajodia [BJ93], pages 544–545.

[RSSS93] RAGHU RAMAKRISHNAN, DIVESH SRIVASTAVA, S. SUDARSHAN AND PRAVEEN SESHADRI. Implementation of the CORAL Deductive Database System. In Buneman and Jajodia [BJ93], pages 167–176.

[SO90] H. C. M. DE SWART AND W. M. J. OPHELDERS. Tableaux, Resolution and Complexity of Formulas. Stichting Mathematisch Centrum, Amsterdam, 1990.

[SO93] H. C. M. DE SWART AND W. M. J. OPHELDERS. Tableaux versus Resolution; a Comparison. *Fundamentae Informaticae*, 18:109–127.

[SS86] L. STERLING AND E. SHAPIRO. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, London, 1986.

[SSW94] KONSTANTINOS SAGONAS, TERRANCE SWIFT AND DAVID S. WARREN. XSB as a Deductive Database. In Snodgrass and Winslett [SW94], pages 442–453.

[ST91] ODED SHMUELI AND SHALOM TSUR. Logical Diagnosis of $\mathcal{LDL}$ Programs. *New Generation Computing*, 9:73–100, 1991.

[SW86] EDWARD SCIORE AND DAVID S. WARREN. Towards an Integrated Database-PROLOG System. In Kerschberg [Ker86], pages 293–306.

[SW94] RICHARD T. SNODGRASS AND MARIANNE WINSLETT, editors. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23-2, Minneapolis, Minnesota, May 1994. ACM Press.

[Swa93] H. C. M. DE SWART. *Logic: Mathematics, Language, Computer Science and Philosophy*, volume 1. Verlag Peter Lang, Frankfurt a. M., 1993.

[Swa94] H. C. M. DE SWART. *Logic: Mathematics, Language, Computer Science and Philosophy*, volume 2. Verlag Peter Lang, Frankfurt a. M., 1994.

[SZ86] DOMENICO SACCÀ AND CARLO ZANIOLO. The Generalized Counting Method for Recursive Logic Queries. In Goos and Hartmanis [GH86], pages 31–53.

[Tar88] A. TARSKI. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1988.

[Top87] RODNEY W. TOPOR. Domain-Independent Formulas and Databases. *Theoretical Computer Science*, 52:281–306, 1987.

[Tsu88] S. TSUR. LDL—a Technology for the Realisation of Tightly Coupled Expert Database Systems. *IEEE Expert*, Fall 1988.

[TZ86] S. TSUR AND C. ZANIOLO. LDL. In WESLEY CHU, GEORGE GARDARIN AND SETSUO OHSUGA, editors, *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, 1986.

[Ull88a] JEFFREY D. ULLMAN. *Principles of Database and Knowledge Base Systems*, volume I. Computer Science Press, Stanford University, 1988.

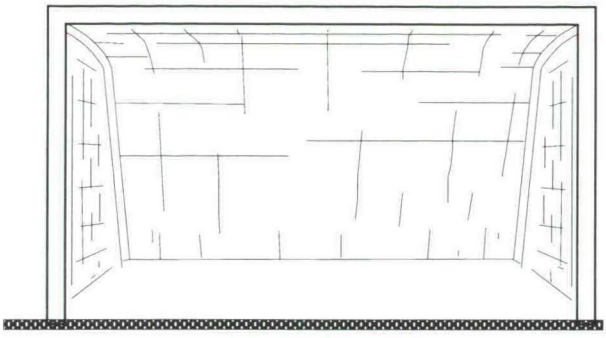
[Ull88b] JEFFREY D. ULLMAN. *Principles of Database and Knowledge Base Systems*, volume II. Computer Science Press, Stanford University, 1988.

[Vie86] LAURENT VIEILLE. Recursive Axioms in Deductive Databases: the Query/Subquery Approach. In Kerschberg [Ker86], pages 253–264.

[Vol87] HUGO VOLKER. Model Theory of Deductive Databases. In E. BÖRGER, H. KLEINE BÜNING AND M. M. RICHTER, editors, *First Workshop on Computer Science Logic, CSL'87*, volume 329 of *Lecture Notes in Computer Science*, pages 322–334, Karlsruhe, FRG, 1987. Springer-Verlag.

[Vol89] HUGO VOLKER. The semantics of Disjunctive Deductive Databases. In E. BÖRGER, H. KLEINE BÜNING AND M. M. RICHTER, editors, *Third Workshop on Computer Science Logic, CSL'89*, volume 440 of *Lecture Notes in Computer Science*, pages 409–421, Kaiserslautern, FRG, 1989. Springer-Verlag.

"*. . . it wanted to update the database . . .*"



Chapter 2

Integrity Constraint Checking

Changes in databases are essential. A database models some part of the universe. Because of the continuous changes in this world, deductive databases which model that world, must also be subject to change. When rules are specified which the database must obey, then each change of the database results in a new database instance, which may conflict with these rules. In this chapter, the handling of such rules, called *semantic integrity constraints*, after such changes, called *updates*, is elaborated.

2.1 Updates

Sooner or later, facts, rules or even constraints change. Such a change is called an *update* to the database. An update is represented by a ground literal in case of a change in the fact base, a rule or a negated rule in case of a change in the rule base, or a constraint or a negated constraint in case of a change in the constraint base. Two kinds of updates are distinguished.

DEFINITION 2.1 Let U be an update to a deductive database D. U is called an *insertion* (resp. a *deletion*) if it is a positive (resp. negative) fact, rule or constraint.

Besides insertions and deletions in a transaction another kind of update is possible, i.e., the replacement. In real database applications, it often happens that data already present in the database has to be modified, because the data was incorrectly added to the database or because only a small part of a fact has to be adjusted in order to comply a change in the real world. For instance, a person's address is changed when this person has removed to another appartment. However, this person is not removed from the database and entered again into the database as a new person, but only the person's old address is removed and a new address is inserted. In order to represent replacements our language should be extended to express them. For now, we represent a relacement as a deletion followed by an insertion.

A change of a database state often consists of a sequence or set of updates.

DEFINITION 2.2 A *transaction* is a set of updates.

Let U be an update to a deductive database D. Then the database resulting from updating D by U is denoted by D_U. Let T be a transaction for the database D. Then the result of executing all updates in T is denoted by D_T. A transaction is *indivisible*, i. e. , all updates in the transaction are executed, while executing only a non-empty subset of T is prohibited. This means that from a database state D after transaction T the next database state is:

- D_T if the transaction is accepted, or
- D if the transaction is rejected.

We call the database D_U, in case of update U, and D_T, in case of transaction T, the *updated database*, while D is often called the *current database*.

For the moment only updates of facts, represented by ground literals, are considered. In 5.2.1 and 5.2.2 updates of rules resp. constraints are elaborated. For the time being, updates are supposed to be fact updates. Fact updates to databases are supposed to go via relations in which those facts are stored. A relation that allows an update is called an *updatable relation*. For instance, the comparison relations in Θ are not updatable. In most cases, updates of derived relations are not allowed. These can only be implicitly updated by explicit updates in the defining relations. This process of *materialization* can cause severe problems. In order to avoid these problems, in the remainder of this thesis a strict distinction between base and derived relations will be made, in other words the database is supposed to be structured. Hence, only base relations are supposed to be updatable. In the literature, several articles describe updates of derived relations and their consequences to the base relations; in relational databases this is also known as the *view updating problem* (see [CM89, Dec90, SW94b]).

Now, two ways of describing changes in deductive databases are studied. First, *induced updates* are described, which give a precise notion of the changes in the database. Second, *potential updates* are described, which give a global notion of the changes in the database.

2.1.1 Induced Updates

In deductive databases, the presence of rules can create complex situations, because an update may cause several other implicit changes to the database, which even depend on the presence of some facts before the update. These changes will be expressed by induced updates, where each induced update may be either an *induced insertion* or an *induced deletion*. The intuitive idea behind induced insertions is that a new instance of the atom in the head of a rule is implied after an update, while it could not be derived in the old database state for that particular rule. One of the causes of the derivation of such a new instance is an insertion in a relation appearing in the body of that rule. Because of the absence of the update in the old state, the rule could not be applied, but after the insertion its application is possible. This intuition is illustrated by the next example.

EXAMPLE 2.1 Let D be a database which contains the following rule:

$$R : a(X) \longleftarrow b(X), B(X),$$

where $B(X)$ is a conjunction of literals in which X is free. Suppose $b(1)$ is an insertion to D and not already present in D, which means that $a(1)$ does not hold in D. By the insertion of $b(1)$ the first part of the body is satisfied. So, the application of R for substitution $\{X/1\}$ is no longer prevented by the absence of $b(1)$. The only condition that has to be fulfilled is that the instantiated remainder of the body of the rule, i.e., $B(1)$, holds in $D_{b(1)}$. When this condition is fulfilled the relation a will be updated by $a(1)$. We say that $a(1)$ is an *induced insertion*, which is *induced by* $b(1)$.

REMARK An induced insertion can also be caused by a deletion in a relation. For instance, when we have a rule $R : a(X) \longleftarrow \neg b(X), B(X)$ and $b(1)$ is present in the database, then its presence prevents the derivation of $a(1)$. When deleting $b(1)$ from the database, $\neg b(1)$ will hold in the updated database. As before, when $B(1)$ holds also in the updated database, the insertion of $a(1)$ is induced.

The intuitive idea behind an induced deletion is that an instance of the atom in the head of a rule can no longer be derived after an update, while it could be derived in the old database state for that particular rule. One of the causes of the fact that the derivation of such a new instance is no longer possible in the updated database is a deletion in a relation appearing in the body of that rule. Because of the presence of the update in the old state, the rule could be applied, but after the deletion its application is no longer possible. This intuition is also illustrated by an example.

EXAMPLE 2.2 Let D be a database that contains the rule of the previous example and contains fact $b(1)$. Suppose $b(1)$ is a deletion from D. This implies that $a(1)$ is deleted from relation a iff $B(1)$ is satisfied in D.

REMARK An induced deletion can also be caused by an insertion in a relation. For instance, when we have a rule $R : a(X) \longleftarrow \neg b(X), B(X)$ and $b(1)$ is absent in the database, then its absence allows the derivation of $a(1)$. When adding $b(1)$ to the database, $\neg b(1)$ will no longer hold in the updated database. As before, when $B(1)$ holds in the old database state, the deletion of $a(1)$ is induced.

Note that in case of induced deletions $B(1)$ has to be evaluated in the old database state. Suppose we evaluated $B(1)$ in the updated database, then the induced deletion $a(1)$ may not be derived, because of the fact that $B(1)$ no longer holds in the updated database.

We describe the concept of induced insertions and deletions more formally:

DEFINITION 2.3 Let U be an update to a database D. Let $R : C \longleftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$ be a deductive rule. Let L be a ground literal, which is unifiable with L_i in R, for some i, and γ be the most general unifier of L and L_i. Let $C' = (C\gamma)\sigma$, where σ is a substitution for which $(L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n)\gamma\sigma$ is true in D_U. Then C' is called *positively directly induced by* L over D_U (with respect to R).

Note that when C' is positively directly induced by L over D_U it holds that if L holds in D_U, then C' holds in D_U.

DEFINITION 2.4 Let U be an update to a database D. Let $R : C \leftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$ be a deductive rule. Let L be a ground literal which is unifiable with the complement of L_i in R, for some i, and γ be the most general unifier of the complement of L and L_i. Let $C' = (C\gamma)\sigma$, where σ is a substitution for which $(L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n)\gamma\sigma$ is true in D. Then $\neg C'$ is called *negatively directly induced by* L over D_U (with respect to R).

When it is clear which update U to which database D is meant we skip the frase "over D_U". Note that from $\neg C'$ is negatively directly induced by L over D_U it follows that if $\neg L$ holds in D, then C' holds in D.

DEFINITION 2.5 Let U be an update to a database D. A literal L' is *directly induced by* a literal L if L' is either positively or negatively directly induced by L.

REMARK Because we assumed that rules are range-restricted, C' in DEFINITION 2.3 and DEFINITION 2.4 is ground.

DEFINITION 2.6 Let U be an update to a database D. A literal is *induced by* L over D_U iff

(i) it is directly induced by L over D_U, or

(ii) it is directly induced by a literal induced by L over D_U.

DEFINITION 2.7 Let D be a deductive database and let U be an update. Each literal induced by update U over D_U and U itself is called an *induced update* with respect to update U (or simply *induced update* if it is clear from the context which update is involved).

DEFINITION 2.8 Let U be an update to a database D. A positive induced update is called an *induced insertion* and a negative induced update is called an *induced deletion*.

From the previous remark it follows that induced updates are ground. Note that induced updates are generated by the application of one particular rule. An induced insertion could already be present in the old database state, because it could be derived by the application of some other rule in the old state. So, the induced insertion does not have to be a real update of the database. In case of an induced deletion, say $\neg A$, where A is a ground atom, the fact A cannot be derived from the given rule, but it is possible that it can be derived by the application of some other rule in the updated database. So, the induced deletion does not have to be a real update of the database either. The next definition makes a distinction between induced updates that are real and induced updates that are not real.

DEFINITION 2.9 Let U be an update to a database D. An induced update is called *effective* if it is an induced insertion which does not hold in D, or, if it is an induced deletion which does not hold in D_U, otherwise, it is called *ineffective*.

REMARK There exists a asymmetry between induced insertions and induced deletions. Suppose U is an update to database D A is some induced insertion with respect to some rule R. Then A holds in D_U because it can be derived by R. However, if $\neg B$ is some induced deletion with respect

to some rule R', then still B may be derived from D_U by some other rule. So, an induced deletion is effective only if there are no derivation paths leading to the negation of the induced deletion in D_U.

DEFINITION 2.10 Let T be a transaction. An effective induced update is called an *explicit update* if it belongs to T. An effective induced update is called an *implicit update* if it is induced by an update U in T.

REMARK An implicit update is an effective induced update which is derived by the application of at least one rule.

Here, the definition of "induced update" is a slight reformulation of the definition of Bry, Manthey and Decker (see [BDM87]). It is even a correction, because they demand that σ is a substitution for which $(L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n)\gamma\sigma$ is true in D_U instead of D in DEFINITION 2.4. This leads to a counterintuitive idea about induced deletions. The next example illustrates this problem, when we assume that their definition is the proper definition for induced deletions.

EXAMPLE 2.3 Let D be a database with the following rule and fact base:

RULES

R_1: $a(X) \longleftarrow b(X, Y), e(Z), \neg c(Y, Z)$

R_2: $c(Y, Z) \longleftarrow d(X, Y), e(Z), \neg b(X, Y)$

FACTS

F_1: $b(1, 2)$

F_2: $d(1, 2)$

F_3: $e(3)$

Note that in this database state $a(1)$ holds. Let the update U be a deletion of the fact $b(1, 2)$. This results in an induced insertion by application of R_2, namely $c(2, 3)$. Further, this update should imply an induced deletion of the derived fact $a(1)$ with respect to the first rule, because the body of R_1 was fulfilled for substitution $\{X/1, Y/2, Z/3\}$ and because of the deletion of $b(1, 2)$ the body of R_1 does not hold anymore for this substitution. However, when the side literals of $b(X, Y)$ for the instantiation $\{X/1, Y/2\}$ are evaluated in D_U, i. e., $e(Z), \neg c(2, Z)$ is evaluated in D_U, we see that it does not hold in D_U. This means that when the induced updates were defined by evaluating the side literals in D_U as in [BDM87], the expected induced update $\neg a(1)$ will not be derived, as a consequence of the simultaneous update of the database by the deletion $b(1, 2)$ and induced insertion $c(2, 3)$. The evaluation of $e(Z), \neg c(2, Z)$ in D instead of D_U corresponds to the proper intuition about induced deletions. For the deletion of $b(1, 2)$ implies that the body of R_1 for substitution $\{X/1, Y/2\}$ does not hold, resulting in the fact that $a(1)$ cannot be derived. This is a change compared to the previous state iff $a(1)$ is derivable in D iff $e(Z), \neg c(2, Z)$ is derivable in D.

Note that by definition, an update is an induced update as well. The reason for this is that an update can be seen as induced by itself, i. e. , without any number of applications of rules the update is presented to the database.

Further, Bry, Manthey and Decker defined the induced updates as effective induced updates, while here an induced update is not necessarily a real update to the database. So, in this thesis the meaning of induced update is different from Bry, Manthey and Decker's meaning of induced update, although the general idea is the same. By using this view of induced updates other concepts are easier to define. The following example illustrates the definitions with respect to induced updates.

EXAMPLE 2.4 Let D be a deductive database with the following rule and fact base:

RULES

R_1: $mother(X, Y) \longleftarrow husband(Z, X), father(Z, Y)$

R_2: $parent(X, Y) \longleftarrow father(X, Y)$

R_3: $parent(X, Y) \longleftarrow mother(X, Y)$

FACTS

F_1: $father(1, 10)$

F_2: $father(1, 11)$

Note that the deducible facts are $parent(1, 10)$ and $parent(1, 11)$. Suppose the update to D is $husband(1, 2)$. The facts $mother(2, 10)$ and $mother(2, 11)$ are (positively) directly induced by $husband(1, 2)$ by using rule R_1 and facts F_1 and F_2. In turn, $mother(2, 10)$ resp. $mother(2, 11)$ (positively) directly induces $parent(2, 10)$ resp. $parent(2, 11)$. So, all induced updates with respect to $husband(1, 2)$ are $husband(1, 2)$ itself, $mother(2, 10)$, $mother(2, 11)$, $parent(2, 10)$ and $parent(2, 11)$.

When a deductive database is updated several induced updates may result. Some of them are ineffective, which means that these induced updates redundantly update the database. This kind of redundancy is illustrated by the following example.

EXAMPLE 2.5 Let D be a database with the facts of EXAMPLE 2.4 and the following rule base:

RULES

R_1: $mother(X, Y) \longleftarrow husband(Z, X), father(Z, Y)$

R_2: $is\_parent(X) \longleftarrow father(X, Y)$

R_3: $is\_parent(X) \longleftarrow mother(X, Y)$

Note that the only difference between this example and EXAMPLE 2.4 is the usage of the unary predicate *is_parent* instead of the binary predicate *parent*. Suppose $father(1, 12)$ is the update to this database. The only induced update with respect to $father(1, 12)$ is $is\_parent(1)$. However,

is_parent(1) is already deducible from the facts *father*(1, 10) and *father*(1, 11) in the old database state.

The definitions in this section can be reformulated for transactions. In this case an update U is considered as an update appearring in a transaction T leading to an updated database D_T. Now, the following definition is permitted.

DEFINITION 2.11 Let T be a transaction to a database D. A literal L is *induced by* T iff it is induced by an update in T. L is called an *induced update* with respect to T (or simply *induced update* if it is clear from the context which transaction is involved).

In the remainder of this thesis we can extend the definitions applied to updates U and updated databases D_U to analoguely defined definitions for an update U in a transaction T and updated databases D_T. Further on in this thesis we use both versions when appropriate.

2.1.2 Potential Updates

In order to make clear which relations are updated without using the facts in the database, *potential updates* are introduced. As for the definition of induced updates some preparations for the definition of potential updates are useful. Analogous to the distinction between induced updates, a potential update is either a *potential insertion* or a *potential deletion*.

DEFINITION 2.12 Let $R : C \longleftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$ be a deductive rule. Let L be a literal which is unifiable with L_i in R, for some i, and γ be the most general unifier of L and L_i. Let $C' = (C\gamma)$. Then we say that C' *positively directly depends on* L with respect to R.

DEFINITION 2.13 Let $R : C \longleftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$ be a deductive rule. Let L be a literal which is unifiable with the complement of L_i in R, for some i, and γ be the most general unifier of the complement of L and L_i. Let $C' = (C\gamma)$. Then we say that $\neg C'$ *negatively directly depends on* L (with respect to R).

Note that from $\neg C'$ negatively directly depends on L it does not follow that if $\neg L$ holds in D, then C' holds in D.

DEFINITION 2.14 A literal L' *directly depends on* a literal L if L' either positively or negatively directly depends on L.

DEFINITION 2.15 A literal *depends on* L iff

(i) it directly depends on L, or

(ii) it directly depends on a literal depending on L.

DEFINITION 2.16 Let D be a deductive database and let U be an update. Each literal depending on update U with respect to some rule of D and U itself is called *a potential update with respect to update* U (or simply *potential update* if it is clear from the context which update is involved).

Note that compared to the definition of positively directly induced and negatively directly induced the state of the database before or after the update is not important. More formally, when in DEFINITION 2.3 and DEFINITION 2.4 substitution σ is not applied potential updates are derived. The definition of potential updates is therefore a slight reformulation of the definition of induced updates.

Note that compared to the definition of induced updates the condition that L is ground is dropped. This means that we no longer are interested in the updates, but only in literals that are influenced by the update. Note further that the update may partially instantiate these literals, which gives a more precise description which part of the corresponding relation is possibly updated.

The definition of "depends on" is also a reformulation of the definition of Bry, Manthey and Decker (see [BDM87]). Here, the relation "depends on" is reflexive. So, any update is by definition also a potential update. As a result an update is an induced update as well as a potential update. So, in this thesis the meaning of potential update is slightly different from Bry, Manthey and Decker's meaning of potential update, although the general idea is the same. As in the case of induced updates, this view of potential updates makes the definition of other concepts easier.

EXAMPLE 2.6 Consider the database with the rules and facts of EXAMPLE 2.4. Suppose the update to this database is $husband(1, 2)$. The literal $mother(2, Y)$ positively directly depends on $husband(1, 2)$ by using rule R_1. In turn, $parent(2, Y)$ positively directly depends on $mother(2, Y)$. So, all potential updates with respect to $husband(1, 2)$ are $husband(1, 2)$ itself, $mother(2, Y)$ and $parent(2, Y)$.

REMARK Each induced update is an instance of some potential update. However, it is possible that some potential update does not have some corresponding induced update as instance. For instance, an update $father(3, 30)$ in the database of EXAMPLE 2.4 would imply a potential update $mother(X, 30)$, for which there exists no instance in the updated database.

DEFINITION 2.17 Let U be an update to a database D. A potential update is called *effective* if the potential update has an instance which is an effective induced update, otherwise, it is called *ineffective*.

REMARK There exists a asymmetry between potential insertions and potential deletions. Suppose U is an update to database D A is some potential insertion with respect to some rule R. Then A holds in D_U if there exist a rule R from which it can be derived. However, if $\neg B$ is some potential deletion with respect to some rule R', then $\neg B$ is derived from D_U iff for each rule in D_U B cannot be derived.

Note that in determining the ineffectiveness of a potential update, all corresponding effective induced updates have to be found. So, by itself the notion of ineffective and effective potential updates has a more theoretical than practical meaning.

Sometimes a potential update is ground, but does not always correspond to an induced update, which is shown by the next example.

EXAMPLE 2.7 Let D be a database with the following rule base and a fact base which is not specified here:

RULES

R_1: $mother(X, Y) \longleftarrow husband(Z, X), father(Z, Y)$

R_2: $natural\_parent(X, Y) \longleftarrow father(X, Y), \neg adopted(X, Y)$

R_3: $natural\_parent(X, Y) \longleftarrow mother(X, Y), \neg adopted(X, Y)$

Let $father(1, 12)$ be an update to D. As a result the potential updates are

$mother(X, 12)$,

$natural\_parent(X, 12)$ and

$natural\_parent(1, 12)$.

As a consequence, the last potential update $natural\_parent(1, 12)$ is also an induced update iff $adopted(1, 12)$ does not hold in the database.

The number of induced updates can outgrow the number of potential updates, as the next example shows.

EXAMPLE 2.8 Suppose in EXAMPLE 2.6 the database consisted of the facts:

$$father(1, 10), father(1, 11), \ldots, father(1, 19).$$

In case of an update $husband(1, 2)$ twenty induced updates appear:

$$mother(2, 10), mother(2, 11), \ldots, mother(2, 19)$$

and

$$parent(2, 10), parent(2, 11), \ldots, parent(2, 19).$$

However, there are only two potential updates

$$mother(2, Y), parent(2, Y).$$

2.2 Integrity Constraints

In order to keep a database from getting into unintended states, some rules are added, which the database has to obey. They comprise a part of the semantics of the database, that is why we refer to these rules as *semantic integrity constraints*, or integrity constraints for short.

DEFINITION 2.18 An *integrity constraint* is a closed first-order formula.

REMARK By allowing negative literals in the head of constraints, we are able to express that some facts are *not* allowed in the database.

A database is called *consistent* if it obeys all its specified integrity constraints, else it is called *inconsistent*. A set of formulas is inconsistent if a contradiction from this set can be derived. The set of integrity constraints itself is supposed to be consistent. For a review of the subject on *validation of a set of integrity constraints* we refer to [BDM87, BM86, BM87, IÖ93, NA86, QS87, QW86, WY92].

REMARK Integrity constraints are intended to express the intended semantics of the database. However, when a database obeys its integrity constraints, the semantics of the database does not change when an integrity constraint is removed from the database. The fact and rule base are still the same; so, without the constraint the same information can be derived from the database.

Each constraint plays a specific role in database integrity. In the remainder of this section, constraints are classified in classes and subclasses, each expressing a specific role. Further, constraints related to the relational model, which we will call *relational constraints*, are classified. Constraints that are not necessarily related to the data model, in this case the relational model, are called *user-defined constraints*.

2.2.1 Static versus Dynamic Constraints

Two major classes of constraints are distinguished, the class of *static constraints* (sometimes called *state constraints* to ephasize that they constrain the database state) and the class of *dynamic constraints*. Static constraints constrain the data in a database state. Dynamic constraints constrain the transitions of one database state into other states. Dynamic constraints constrain the transition of one state to another, which means that two states are involved. Therefore, these constraints are also known as *transition constraints*.

EXAMPLE 2.9 The constraint

> "*The salary of an employee can be raised by a maximum of 1000 pounds*"

is dynamic, because when an employee's salary is raised one has to know the employee's salary before the raise in order to check the constraint. The constraint

> "*The salary of an employee cannot exceed 8000 pounds*"

is obviously static. One can check this constraint without any knowledge of the previous database state.

REMARK The consistency of a database state is solely determined by its static constraints. One cannot speak about the consistency of one single database state with respect to the dynamic constraints. Then, at least two database states must be involved.

This remark plays an important role in the definition of *proper updates*.

DEFINITION 2.19 A *proper* database update is an update that transforms some consistent database state into some other consistent database state without violating any dynamic constraint.

In this thesis, only static constraints are considered. So, by assumption a proper update will not violate any dynamic constraint.

2.2.2 Relational Constraints versus General Constraints

In the class of static constraints several subclasses are distinguished. For instance, the relational constraints are classified by the number of relations, attributes and tuple variables involved in expressing the constraint. Let D be a database consisting of the relations $R_1, R_2, \ldots, R_n$. The general form of a static constraint for D is:

$$Q_1\bar{t_1} \in R_{i_1}\ Q_2\bar{t_2} \in R_{i_2}\ \ldots\ Q_k\bar{t_k} \in R_{i_k}\ [\lambda(A_1, A_2, \ldots, A_l, \bar{t_1}, \bar{t_2}, \ldots, \bar{t_k})].$$

In this formula:

(i) Q_j is the quantifier $\forall$ or $\exists$,

(ii) $\bar{t_j}$ is a tuple variable for relation R_{i_j},

(iii) $A_1, A_2, \ldots, A_l$ are attributes selected from the set $\bigcup_{r=1}^{k} \mathcal{A}_i$

where $R_{i_j} \in \{R_1, R_2, \ldots, R_n\}$, $j = 1, 2, \ldots, k$, $\mathcal{A}_i$ is the set of all attributes in R_i for $i = 1, 2, \ldots, n$ and λ is an expression in tuple variables, which uses attributes $A_1, A_2, \ldots, A_l$, comparison operators, logical connectives, set constructors and arithmetic expressions. Let Θ be the set of comparison relations. The classification of constraints in Table 2.1 is in the first place determined by the number m of distinct relations among the relations $R_{i_1}, R_{i_2}, \ldots, R_{i_k}$ and in the second place by the number l of attributes appearing in expression λ. Let $m = |\{R_{i_1}, R_{i_2}, \ldots, R_{i_k}\}|$, where $|.|$ operates on sets and gives the number of elements of its argument, which is a set. In Table 2.1 the symbol "-" in a column states that the value of the corresponding parameter can be chosen without restriction. Further, $\mathcal{D}$ and $\mathcal{D}_i$ represent domains and $\mathcal{B}_i$ represents some subset of attributes of the relation that is involved, for each i.

Each of the constraints in Table 2.1 plays a particular role in the relational model. For instance, *tuple constraints* constrain the combination of elements in a tuple. Two special kinds of tuple constraints are *domain constraints*, sometimes called *type constraints*, which constrain only one attribute in a tuple, and *interdomain constraints*, which constrain more than one attribute in a tuple.

The *dependencies* on relations constrain relations by demanding equality or just inequality of values in several arguments of relations. Dependencies play an important role in the way a relational database is set up and maintained. One of the most important functions of dependencies is that they determine the decomposition of tables in order to get the relational schema in some kind of normalized form (see [Kob85, Ull88]). Some well known dependencies are the *functional dependency*, the *multivalued dependency*, which is a variant of a *template dependency*, and the

Kind of Constraint	m	k	Quantification	l	λ
TUPLE CONSTRAINT	1	≥ 1	$Q_1 = \forall$	–	$\lambda(A_1, A_2, \ldots, A_l, \bar{t})$
Domain Constraint	1	1	$Q_1 = \forall$	1	$A(\bar{t}) \subset \mathcal{D}, A \in \mathcal{A}_1$
Interdomain Constraint	1	1	$Q_1 = \forall$	≥ 2	$A_1(\bar{t}) \subset \mathcal{D}_1, \ldots, A_r(\bar{t}) \subset \mathcal{D}_r$
Permanent Functional Dependencies	1	1	$Q_1 = \forall$	2	$A_1(\bar{t}) = f(A_2(\bar{t}), \ldots, A_r(\bar{t}))$
Theta Dependencies	1	1	$Q_1 = \forall$	2	$A_1(\bar{t})\,\theta\,A_2(\bar{t}), \theta \in \Theta, \{A_1, A_2\} \subset \mathcal{A}_1$
DEPENDENCIES	1	≥ 2	$Q_1 = \forall$, $Q_r \in \{\forall, \exists\}$, $r \geq 2$	–	–
Functional Dependencies	1	2	$Q_1 = \forall$, $Q_2 = \forall$	≥ 2	$\mathcal{B}_1(\bar{t}_1) = \mathcal{B}_1(\bar{t}_2) \rightarrow \mathcal{B}_2(\bar{t}_1) = \mathcal{B}_2(\bar{t}_2)$
(Primary/Candidate) Key	1	2	$Q_1 = \forall$, $Q_2 = \forall$	≥ 2	as functional dependencies, where $\mathcal{B}_2 = \mathcal{A}_1$
Key Dependencies	1	2	$Q_1 = \forall$, $Q_2 = \forall$	≥ 2	as functional dependencies, where $\mathcal{B}_2 = \mathcal{A}_1 - \mathcal{B}_1$
Template Dependencies	1	≥ 2	$Q_1 = \cdots = Q_{r-1} = \forall$, $Q_r = \exists$	–	–
Multivalued Dependencies	1	3	$Q_1 = Q_2 = \forall$, $Q_3 = \exists$	≥ 2	$\mathcal{B}_1(\bar{t}_1) = \mathcal{B}_1(\bar{t}_2) \rightarrow (\mathcal{B}_1 \cup \mathcal{B}_2)(\bar{t}_1) = (\mathcal{B}_1 \cup \mathcal{B}_2)(\bar{t}_3) \wedge (\mathcal{A}_2 - \mathcal{B}_2)(\bar{t}_2) = (\mathcal{A}_2 - \mathcal{B}_2)(\bar{t}_3)$
INTERRELATION CONSTRAINTS	≥ 2	≥ 2	–	–	–
Into Constraints	2	2	$Q_1 = \forall$, $Q_2 = \exists$	–	$\mathcal{B}(\bar{t}_1) = \mathcal{B}(\bar{t}_2)$, where $\mathcal{B}$ is a subset of a candidate key of R_2
Onto Constraints	2	2	$Q_1 = \forall$, $Q_2 = \exists$	–	$\mathcal{B}(\bar{t}_1) = \mathcal{B}(\bar{t}_2)$, where $\mathcal{B}$ is a subset of a candidate key of R_1
EXISTENTIAL CONSTRAINTS	≥ 1	≥ 1	$Q_1 = \exists$, $Q_r \in \{\forall, \exists\}$, $r \geq 2$	–	–

Table 2.1: Classifying relational constraints

implicational dependency, which are all checked by using their syntactical properties. Template dependencies are further subdivided in other subclasses, such as the classes of *embedded multivalued dependencies*, *embedded* resp. *generalized mutual dependencies*, *join dependencies* and *subset dependencies* (see [Kob85]). The most famous functional dependency is the *key dependency*. For a unique identification of tuples and the possibility of indexing techniques key dependencies are introduced. A *key* for a relation R is a subset of attributes of R such that its values uniquely identify the tuples of R. Key dependencies with some other additional properties lead to other specialized dependencies such as *prime dependencies*, *transitive dependencies*, *(pseudo)partial dependencies*, and *(pseudo)reflexive dependencies* (see [Ull88]).

Not only constraints within one particular relation are thinkable, i. e. , $m = 1$, but it is also possible that one relation constrains the other in one or more attributes. Examples of such *interrelation constraints* are the *into constraints* and the *onto constraints*.

All previously mentioned constraints are ruled by a universal quantifier. However, constraints may be ruled by an existential quantifier in order to guarantee that a certain value exists in one or more relations. These constraints are called *existential constraints*.

Sometimes it is necessary to state some constraints on a relation or several relations, where all tuples in a relation are involved; for example, the values of all tuples on some attribute set or the number of tuples of a column or several columns must fulfill some constraint. For instance, the average or sum of the values is not allowed to exceed some prescibed value, the number of tuples in a relation must be less than some maximum limit, etc.. The class that contains all these kinds of constraints is known as the class of *aggregate constraints*. More on relational constraints can be found in [Kob85, Ull88]. This classification is certainly not complete, and does not pretend to be, but it shows the variety of constraints and the importance of constraints in the relational model. In [Das90] a global classification is given for constraints in deductive databases.

In relational database systems, several classes of these constraints can be handled automatically (see [Bla81, Bro78, Del87, FW83, GA93, LR84, Qia88a, Qia88b, SK88, Sto75, SV85, WSK83]. However, the user does not have the possibility to define arbitrary constraints, which are also supported by those systems. When considering the consistency of a database management system, two consistency types are distinguished, namely the *internal consistency* and the *external consistency* of the system. In order to maintain the internal consistency the system should allow the specification of constraints to maintain the *data model consistency* and the *database schema consistency*, such as domain constraints and keys. The external consistency related to the database instance instead of the database schema is the consistency we are interested in in this thesis. It should be offered by the system by allowing the specification and the verification or maintenance of *integrity constraints*.

When considering deductive database systems, or other deductive systems that are either based on extensions of the relational model or a completely diffent data model, one wants to have the

opportunity to specify constraints on base relations as well as derived relations for the maintenance of the external consistency of the system. The user may even want to have the freedom to express more general constraints, such as *temporal constraints* and *deontic constraints*. Temporal constraints also contain a notion of time. It contains concepts like *during*, *before*, *next*, *starts*, *finishes*, etc.. More on temporal constraints can be found in [Kun85] and [Ple93]. In [Ple93] integrity checking in temporal deductive databases is described. In [Kun85] temporal constraints are checked by using a tableaux based proof procedure. Deontic languages contain other language constructs in order to be able to express more advanced constraints. Sometimes a constraint is not persistent, for instance, a constraint may be interpreted as "it should be desirable that this constraint holds, but not obligated". In deontic logic this aspect of constraint specification is elaborated. More on deontic constraints can be found in [MWW89], [WMW89] and [Kwa91]. In these papers on temporal and deontic constraints, attention is paid to the *description* of more general constraints in a temporal or deontic language respectivily. Further, one can also specify *fuzzy constraints*, which are constraints with fuzzy knowledge, containing concepts like *many*, *most*, *almost*, *some*, etc.. In [RM88] fuzzy constraints are elaborated in the relational case. In this thesis, the checking of temporal, deontic and fuzzy constraints are not elaborated, although in my opinion most of the results of this thesis can be used to automate the checking of those constraints as well.

In this thesis, attention is paid to the efficiency of *checking* the constraints. This is done in a deductive database setting. Hereby, we restrict ourselves to only a small class of constraints. As the definition of integrity constraint showed, we restrict ourselves to first-order formulas, but for the moment, we consider only a subclass of these formulas, namely the *universally quantified* formulas. These formulas have the following form:

$$\forall X_1 \cdots \forall X_n [L_1 \wedge \cdots \wedge L_m \rightarrow Q],$$

or equivalently:

$$\neg(\exists X_1 \cdots \exists X_n [L_1 \wedge \cdots \wedge L_m \wedge (\neg Q)]),$$

where $m, n \geq 0$, each L_i is a literal, each variable X_i, $i = 1, 2, \ldots, n$, occurs in one or more L_j, $j = 1, 2, \ldots, m$, and Q is a literal.

In general, we not only want complete freedom in expressing all kinds of constraints on the data, but we also want the system to be responsible for checking the constraints. Logical database systems, especially deductive database systems, should offer this functionality and this thesis intends to show that they can. However, it turns out that the automated checking of such constraints causes a lot of complications. These complications are elaborated in the next chapter. First some assumptions on the interaction between updates and integrity constraints have to be made.

2.3 Updates and Integrity Constraints

Before checking integrity constraints after an update or transaction for a particular database state, some agreement must exist about transformations and integrity checking. For instance, a decision

has to be made at which moments during the transaction the consistency of the database is checked and what actions have to be taken when an inconsistency is found. The starting points and basic assumptions concerning integrity constraint checking are made clear in this section.

2.3.1 Immediate versus Deferred Constraints

In general, constraints can be *immediate* or *deferred*. A constraint is called *immediate* if it is satisfied during and after the execution of the transaction. A constraint is called *deferred* if it is satisfied after the execution of the transaction. In this thesis, a transaction is supposed to be executed as a primitive action, i. e. , it is not possible that constraints are violated during the transaction. Therefore, constraints are only checked after a transaction has been executed. So, here constraints are supposed to be deferred.

2.3.2 Checking versus Maintenance

Another issue is how the integrity of the database is restored after an update leading to an inconsistent database. This is the issue of *integrity maintenance*. Maintaining the integrity of the database is done by *enforcement* of each integrity constraint of the database. For instance, let there be a database containing the following rules:

> "*each student who passes all final exams will graduate to the next grade*"

and

> "*a student who did not pass all final exams must do the grade over again.*"

Suppose some constraint is specified for this database, which states that *each classroom can contain at most thirty students*. Suppose the database represents thirty students, where each student is in the second grade. Suppose further that the database represents the facts *John is in the third grade* and *each of the thirty students in the second grade did pass all final exams*. The insertion into this deductive database of the fact that *John did not pass his final exam for history*, does affect the integrity constraint. For, the rules applied to the facts and the update implies that the third grade consists of *John* and *all thirty graduated students of the second grade*. Note that this integrity constraint can be enforced by splitting the third grade into two groups of fifteen and sixteen students respectively, each group having one separate classroom, provided that this is not forbidden by any other constraint.

This restoration of the consistency of a database state is the task of a deductive maintenance system. However, the first step in maintaining the consistency of a database is the use of some integrity checking system, which only task is the signalling of an inconsistency. In case of an inconsistency, after a check some information about facts, rules and constraints that are involved in the inconsistency should be available. A maintenance system should determine the cause the inconsistency. It must reason with some meta-constraints in order to justify why some facts, rules or constraints, which are involved in the inconsistency, are more likely to hold than others. Also, some or all of these decisions can be made interactively by the user.

REMARK Deductive rules can be used to automatically update a database, i.e., a kind of integrity maintenance. For instance, $parent(X, Y) \longleftarrow child(Y, X)$ automatically updates the relation *parent* when an update in the relation *child* takes place. In fact, a constraint

$$\forall X \forall Y [child(Y, X) \rightarrow parent(X, Y)]$$

demands the same. However, in this case we can choose the way the consistency of the database is corrected after an inconsistency, i.e., accept the update in the relation *child* by adding a proper *parent* fact or reject the update in the relation *child*.

In this thesis, we are not interested in how to make the database consistent again but we are only interested in the issue of *integrity checking*. This means that we are only interested in how to detect in the most efficient manner, when the database reaches an inconsistent state. When an update causes an inconsistent database state, we say that the update is not allowed and therefore rejected, which means that the old database state is reached again. So, the consistency is restored by going back to the consistent database state before the update.

2.3.3 Strong versus Soft Constraints

In the previous section, the checking or maintenance of integrity constraints was a feauture of the database as a whole. However, a mixture of checking or maintaining the set of integrity constraints is also possible by specifying for each integrity constraint how important the satisfaction of such a constaint is. Hence, we distinguish strong, soft and selfcorrecting constraints. When a *strong constraint* is violated, the transaction which causes that violation results in an immediate rollback of the transaction. When a *soft constraint* is violated, the user is informed about this violation, an the user is responsible for any action. When a *selfcorrecting constraint* is violated, some action is automatically taken to correct the inconsistency. In a pure integrity maintenance system all constraints would be selfcorrecting. In this thesis, where we are only interested in checking the integrity constraints, all constraints are handled as strong constraints.

2.3.4 Theoremhood versus Consistency View

Integrity constraint checking can be looked at from two perspectives. The first one is called the *theoremhood view* of constraint checking and the second one is called the *consistency view* of constraint checking.

In the first view, the database is called *consistent* when the integrity constraints are seen as formulas which logically follow from the database. In the second view, the database and integrity constraints are seen as one logical program which must be consistent. In general these two views are not equivalent. The consistency view gives a weaker notion of consistency in databases than the theoremhood view, because in the theoremhood view $DB \models IC$ must be satisfied, i.e., every model of DB must be a model of IC, while in the consistency view the integrity constraints and the database are one, so, DB and IC have to be satisfied by only one model. Therefore the consistency view is implied by the theoremhood view. These two views are equivalent if the database

has just one model, for instance in the case of a stratified database. Model theoretically, both views comprise the semantics given to integrity constraints, because databases are supposed to be stratified in this thesis.

Proof theoretically, the integrity of the database is preserved when all the integrity constraints are still derivable from the first order theory representing the new database state.

2.3.5 Inconsistency Indicator versus Integrity Constraint

As we have seen, integrity constraints are an indispensible part of a database. But instead of using integrity constraints in order to monitor the consistency of the database, in this thesis, mainly inconsistency indicators are used. An inconsistency indicator is a statement that holds, when the database becomes inconsistent by a transaction. An inconsistency indicator is the negation of an integrity constraint.

For example, when the constraint

"*each classroom can contain at most thirty students*"

is true in the world we are modelling, then it is reformulated as the inconsistency indicator:

"*there exists a classroom that contains more than thirty students.*"

As stated earlier, the system only deals with universally quantified rules and integrity constraints. Note that the integrity constraint is universally quantified, for it is possible to reformulate the constraint as:

"*for all classrooms it holds that they can contain at most thirty students.*"

As a consequence the inconsistency indicator is existentially quantified. Inconsistency is no longer conceived as a violation of the integrity constraints but as a true evaluation of an inconsistency indicator in the updated database state.

More formally, constraints are expressed as $\neg F$, where F is some closed existentially quantified formula expressed in the underlying language of the theory. So, the indicators are closed as well.

In this thesis, the theory is built by using the inconsistency indicator F instead of the integrity constraint $\neg F$. F is called the *inconsistency indicator* with respect to the constraint. In this thesis, our constraints are supposed to be range-restricted. Note that the range-restrictness property in case of inconsistency indicators means that each variable in a negative literal of an inconsistency indicator must also appear in a positive literal of the inconsistency indicator.

PROPOSITION 2 *If an integrity constraint is range-restricted the corresponding inconsistency indicator is range-restricted.*

Proof: Let $IC : \forall X_1 \cdots \forall X_n[L_1 \wedge \cdots \wedge L_m \rightarrow Q]$ a range-restricted integrity constraint and $II : \exists X_1 \cdots \exists X_n[L_1 \wedge \cdots \wedge L_m \wedge (\neg Q)]$ the corresponding inconsistency indicator, where $m, n \geq 0$, each L_j is a literal, each variable X_i, $i = 1, 2, \ldots, n$, occurs in one or more L_j, $j = 1, 2, \ldots, m$, and Q is a literal. Note that the condition that variables in negative literals of $L_1, L_2, \ldots, L_m$ in II must appear in the positive ones in $L_1, L_2, \ldots, L_m$ in II is fullfilled. Suppose, $\neg Q$ is a positive literal. Then the proposition is proven. Suppose, $\neg Q$ is a negative literal. Note that the variables in Q are all present in one or more positive literals of $L_1, L_2, \ldots, L_m$, because of the range-restrictedness of IC. So, in this case the proposition is also proven. ❑

REMARK The opposite of this proposition may not hold, i. e., a range-restricted inconsistency indicator may have a corresponding integrity constraint which is not range-restricted. For instance, the inconsistency indicator

$$II : \exists X \exists Y[parent(X, Y), student(X)]$$

is range-restricted for obvious reasons, but the corresponding constraint

$$IC : \forall X \forall Y[student(X) \rightarrow \neg parent(X, Y)]$$

is not range-restricted, because variable Y appears in the head of the IC and not in the body of IC.

Note that the proof of this proposition shows that the opposite of the implication holds when Q is positive and is chosen as the literal to appear in the head of the constraint. So, when considering only a subclass of our class of constraints, namely the constraints of which the head literal is positive, the equivalence of both formulas from the range-restrictness point of view is guaranteed. However, because constraints are the most common formulation of integrity, a set of range-restricted constraints is transformed to a set of inconsistency indicators instead of conversely. So, the range-restrictness property is preserved for inconsistency indicators. However, when starting with a range-restricted inconsistency indicator, one must keep in mind that the indicator may represent an integrity constraint, which is not range-restricted. The next example shows that a (range-restricted) inconsistency indicator may represent several (range-restricted) integrity constraints.

EXAMPLE 2.10 Let II represent the following inconsistency indicator:

$$\exists X \exists Y[parent(X, Y), \neg dependent(Y, X), \neg employed(Y), \neg student(Y)]$$

Note that II is range-restricted. Then each of the following range-restricted constraints has II as its corresponding inconsistency indicator:

$$\forall X \forall Y[parent(X, Y), \neg dependent(Y, X), \neg employed(Y) \rightarrow student(Y)]$$

$$\forall X \forall Y[parent(X, Y), \neg dependent(Y, X), \neg student(Y) \rightarrow employed(Y)]$$

$$\forall X \forall Y[parent(X, Y), \neg student(Y), \neg employed(Y) \rightarrow dependent(Y, X)]$$

In a logical sense, all three constraints are equivalent. However, suppose *parent*(1, 2) was the only fact of the database, then we could say that the first constraint does not hold, because of the absence of *student*(2), the second constraint does not hold, because of the absence of *employed*(2), and the third constraint does not hold because of the absence of *dependent*(2, 1). So, when looking at the cause of an inconsistency the constraints may have different meanings. Note that the inconsistency indicator covers either of these situations. Therefore, the cause of an inconsistency of II is at best expressed by an equivalent formula:

$$\forall X \forall Y [parent(X, Y) \rightarrow student(Y) \vee employed(Y) \vee dependent(Y, X)].$$

So, in this sense inconsistency indicators are more powerful and generic than constraints. However, in the maintenance of constraints and indicators their meaning may differ. In this thesis, constraints and inconsistency indicators are viewed from a logical perspective in order to overcome this problem. Besides, it is a problem concerning the maintenance of constraints, which is not elaborated in this thesis.

The relation between constraints and indicators with respect to the consistency of the database is expressed by the following proposition.

PROPOSITION 3 *A database D is consistent with its specified constraints iff all the related inconsistency indicators are false in the database.*

Proof: The database is consistent with its specified constraints iff all constraints are true in D. Because the inconsistency indicators are the negations of the constraints, the lemma easily follows. ❑

In the remainder of this thesis, when the consistency of a database is concerned, inconsistency indicators play a dominant role. The proposed method for integrity constraint checking, which is discussed further on in this thesis, is easier to describe using the concept of inconsistency indicator than using the concept of integrity constraint. Changing these concepts at a later stage could lead to a thesis which is harder to understand and which would lead to several reformulations of definitions and propositions.

2.4 Integrity Constraint Checking in Databases

Now we have introduced updates and inconsistency indicators for databases, a way must be found to check whether a database is consistent after an update or not. In order to determine if an updated database is consistent one could check all inconsistency indicators. This is in most cases a time consuming task.

A full check of all indicators has to be avoided. Before presenting the proper way in order to reach this, some remarks have to be made. When updating a database some derived update, induced or potential, may influence an inconsistency indicator, while it has nothing to do with others. Even,

some derived updates may not influence any inconsistency indicator. This situation is described more formally in the next definition by the concept of *relevant*.

DEFINITION 2.20 Let D be a database and U an update. An induced update (resp. a potential update) is called *relevant* for the integrity check of the updated database D_U, or just *relevant* when it is clear which D and U is meant, if it is relevant to some inconsistency indicator, else it is called *irrelevant*.

DEFINITION 2.21 Let D be a database and II be an inconsistency indicator. II is *influenced by* an update U (resp. a transaction T) if there exists an induced update with respect to U (resp. T) which is relevant to II.

DEFINITION 2.22 Let D be a database and U an update. An inconsistency indicator is called *relevant* for the integrity check in the updated database D_U, or just *relevant* when it is clear which D and U is meant, if there exists an induced update (resp. a potential update) to which it is relevant, else it is called *irrelevant*.

From DEFINITION 2.20 it follows that any induced update, that is an instance of some relevant potential update, is also relevant. Analogue, any induced update, that is an instance of some irrelevant potential update, is also irrelevant.

In methods for checking integrity constraints automatically the full check of inconsistency indicators is prevented by the following assumption:

before an update a database is supposed to be consistent.

This assumption guarantees that after an update to a database

- ➢ only the induced updates can be responsible for an inconsistent state,
- ➢ only inconsistency indicators which are relevant to some induced update have to be evaluated.

REMARK This last item is not true when inconsistency indicators are not domain independent, because then an update could introduce a new symbol, which after its introduction becomes relevant for each inconsistency indicator even those which are irrelevant to any induced update. So, in that case these inconsistency indicators also have to be checked.

The assumption leads to an even stronger restriction on the set of induced updates and indicators, i. e., the previous two restrictions can be refined further to

- ➢ only induced updates that are effective can be responsible for an inconsistent state,
- ➢ only a changed part of an integrity constraint, instead of the full constraint, has to be checked.

The first refinement states that the facts that were already present in the database, thus including all ineffective induced updates, cannot be responsible for any inconsistency in the updated database. This follows directly from the assumption. The second refinement states that only an updated part of the inconsistency indicator has to be evaluated. The part of the inconsistency indicators, which is not updated, does not have to be evaluated; this also follows directly from the assumption. These refinements are described more formally further on in this section.

The assumption mentioned above is used in the method for checking the consistency in deductive databases proposed in this thesis, in order to be able to reach a high degree of efficiency.

In the remainder of this section integrity constraint checking in relational and deductive databases is elaborated further. In the case of deductive databases, two elementary methods are described, i. e. , an integrity checking method based on induced updates and one based on potential updates.

2.4.1 Integrity Constraint Checking in Relational Databases

In this section, integrity checking in relational databases is described. In relational databases no rules exist. So, in relational databases an update cannot cause any induced update. Hence, when the relational database is updated, only the update itself can violate a constraint. When an update influences some inconsistency indicator, this inconsistency indicator is instantiated by the update.

DEFINITION 2.23 Let U be an update and II an inconsistency indicator relevant to U. The simplified instance of II with respect to U leaving out all the occurrences of the update U is now called an *update instance* of II with respect to U.

In fact, only "simplified instances" (terminology of [BDM87]) of only relevant indicators are sufficient to represent the check (see PROPOSITION 4). So, the collection of inconsistency indicators which have to be checked can often be reduced considerably. This simplification of integrity constraints is often called *specialized integrity checking* or *incrementally checking of integrity constraints*.

REMARK The reason for leaving out the occurrences of the update in the update instances is that the update instances are evaluated in the updated database. So, because we know beforehand that the update is always true in the updated database, the update does not have to be evaluated in the updated database.

Update instances play an important role in checking the consistency of a relational database, as the following proposition shows.

PROPOSITION 4 *Let D be a relational database and let U be an update. Suppose D is consistent. Then D_U is consistent iff all update instances of inconsistency indicators with respect to U are false in D_U.*

Proof:
$\Rightarrow$ Suppose D_U is consistent; then all inconsistency indicators are false in D_U (See PROPOSITION 3). D_U can therefore be seen as an interpretation in which all inconsistency indicators are false. Suppose there is an update instance I of some indicator II, which holds in the interpretation D_U. Because I is an instantiation implied by U and $U \in D_U = D \cup \{U\}$, II should also be true in the interpretation D_U. So, a contradiction is derived. Therefore, there is no update instance true in D_U. (Note that in proving this part of the proposition, the assumption that D is consistent) is not used.

$\Leftarrow$ Suppose all update instances with respect to U are false in D_U. In order to prove that D_U is consistent, we have to prove that all inconsistency indicators are false in interpretation D_U. First the collection of all inconsistency indicators, say I, is divided into two disjunct sets:

- I_U; the set of all inconsistency indicators relevant to U, and
- $I_U^C = I / I_U$; the set of all inconsistency indicators not relevant to U.

Let $I \in I_U^C$. Because of the assumption that D is consistent, I is false in D. Because I is not relevant to U, only the facts in D are involved in satisfying I in the interpretation D_U. So, because of the assumption that D is consistent, i. e. , I is false in D, this implies that I is false in D_U. Now, let $I \in I_U$. Let L be a literal of I, which is unifiable with U. Let σ_1 be a most general unifier of U and L. We have to prove that I is false in D_U. Suppose I is true in D_U. Because I is false in D, $I\sigma_1$ must be true in D_U. But $I\sigma_1$ is an update instance of I with respect to U. So, by hypothesis, $I\sigma_1$ is false in D_U. So, we have a contradiction. ❏

REMARK In [Nic79] and [Nic82] Nicolas proved a similar result. However, by constraining ourselves to update instances of inconsistency indicators and by using the definitions and concepts presented in this thesis, the proof could be shortened considerably.

EXAMPLE 2.11 Let D be a relational database consisting of the following facts:

$p(b)$	$q(b, b)$
$p(c)$	$q(b, c)$
$p(e)$	$q(c, a)$

Let $II = \exists X \exists Y[q(X, Y) \wedge \neg p(X)]$ be the only inconsistency indicator specified for D. It is easy to check that D is consistent. Let $q(c, b)$ be an update to D. The update instance that is derived and which has to be false in D_U is $\neg p(c)$. Now, $\neg p(c)$ is false because $p(c)$ is true in D_U. So the update is allowed. Now, suppose the update was the deletion of $p(b)$, then the update instance of II is $\exists Y[q(b, Y)]$. This instance is true in D_U. So the deletion of $p(b)$ is not allowed. Note that an insertion $p(a)$ does not influence the inconsistency indicator and is therefore allowed.

When the whole inconsistency indicator is evaluated instead of some instance of it then subsequently we must instantiate II by using each q(_,_)-fact in D_U and evaluate each of these instances. In cases of a large number of such facts the evaluation could take a lot more time than the sole evaluation of the update instance.

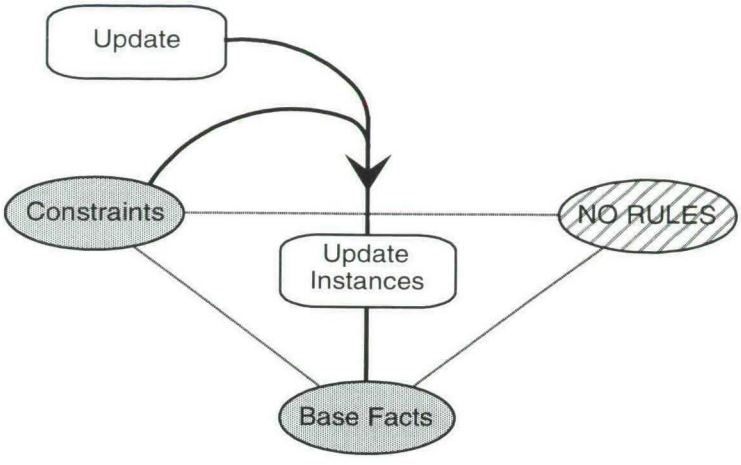


Figure 2.1: Overview of integrity checking in the relational case

PROPOSITION 4 is illustrated in FIGURE 2.1. This figure shows that from the update and the inconsistency indicators some update instances are derived. Further, the update instances are evaluated in the database, which is represented by the triangle of which inconsistency indicators, base facts and an empty rule base are the components. PROPOSITION 4 is also illustrated by EXAMPLE 2.11.

PROPOSITION 4 can easily be generalized from one update to a transaction. Let D be a database and let T be a transaction, then in order to be able to conclude that D_T is consistent, all update instances for each update in a transaction T must be false in D_T.

Note that we can look at a relational database as a deductive database without rules. Note also that in the relational case an update leads directly to instantiations of inconsistency indicators. Note also that from an update and the specified inconsistency indicators a set of update instances can be generated which is possibly empty in the case of an update which is not relevant to any indicator.

2.4.2 Integrity Constraint Checking in Deductive Databases

In case of deductive databases several methods for checking the integrity of a database are available. Compared to the relational case, the deductive case is more complicated, because now not only the update may be responsible for an inconsistent state of the database but other induced updates as well. In this section two elementary methods are presented, each detecting the effective

and relevant induced updates in their own way. The first method is based on the usage of induced updates and the second one is based on the usage of potential updates.

2.4.2.1 Integrity Constraint Checking based on Induced Updates

The method described in this section tries to instantiate inconsistency indicators by using induced updates. It turns out that checking the consistency of the database is done by only evaluating these instantiated inconsistency indicators, which will be called *induced instances*.

DEFINITION 2.24 Let U be an update. Let II be an inconsistency indicator and let L be an induced update relevant to II. The simplified instance of II with respect to L, leaving out all the occurrences of the induced update L, is now called an *induced instance* of II with respect to L.

Note that induced updates are left out from the definition of induced instances because we know beforehand that these updates are always present in the updated database. Checking whether an induced update is in the updated database is redundant.

In the method based on induced updates three phases are distinguished:

(i) *the generation phase*, in which the induced updates are generated,

(ii) *the selection phase*, in which all induced updates relevant with respect to the inconsistency indicators are selected,

(iii) *the evaluation phase*, in which the induced instances of the inconsistency indicators, which were derived in the previous phase, are evaluated.

These phases will be illustrated by using the next example.

EXAMPLE 2.12 Suppose we have the situation as in EXAMPLE 2.4. Suppose we also have an inconsistency indicator which expresses that in this database a person cannot be a *parent* and a student at the same time:

$$II_1 : \exists X \exists Y [parent(X, Y), student(X)]$$

Note that the database was consistent before the update, because $student(1)$ did not hold in the database. In EXAMPLE 2.4 the first phase of the method based on induced updates for this specific database is illustrated: given the update $husband(1, 2)$ the induced insertions $husband(1, 2)$, $mother(2, 10)$, $mother(2, 11)$, $parent(2, 10)$ and $parent(2, 11)$ are found.
In the second phase, from the induced updates all relevant induced updates are selected, i.e., $parent(2, 10)$ and $parent(2, 11)$. For all other induced updates it holds that they are not relevant to any inconsistency indicator in the database. In the third phase, first the induced instances of II_1 are derived from each induced update relevant to II_1, i.e., $parent(2, 10)$ and $parent(2, 11)$. These instances are just the simplified instances of II_1 with respect to $parent(2, 10)$ resp. $parent(2, 11)$, i.e., $parent(2, 10)$, $student(2)$ resp. $parent(2, 11)$, $student(2)$. Second, when evaluating the induced instances of II_1, we find that the induced instances do not hold in the updated database, because of the absence of fact $student(2)$ in the updated database.

REMARK The update *husband*(1, 2) implied several induced updates. However, only a few of them are relevant for checking the consistency of the database. In order to find the induced instances of the inconsistency indicators, it may happen that a large number of ineffective induced updates have to be derived. Note that in the previous example *mother*(2, 10) and *mother*(2, 11) are irrelevant, but were indispensible for deriving *parent*(2, 10) and *parent*(2, 11), which are relevant.

The next proposition justifies that it is sufficient to check only the induced instances of inconsistency indicators for checking the consistency of the updated database.

PROPOSITION 5 *Let D be a consistent database and let U be an update. Then D_U is consistent iff each induced instance of an inconsistency indicator is false in D_U.*

Proof: This property follows from PROPOSITION 4 by reduction to the relational case. Consider the canonical interpretation of D as a relational database. A canonical interpretation consists of true atoms corresponding to the facts which are in the database or derivable from the database by its rules. A unique canonical interpretation can be determined by demanding that the rules are stratified in the sense of [ABW88]. In this thesis, D is supposed to be stratified. Further, treat the induced updates as explicit updates to this database. ❑

PROPOSITION 5 can easily be generalized from one update to a transaction.

PROPOSITION 6 *Let D be a consistent database and let T be an update. Then D_T is consistent iff each induced instance of an inconsistency indicator is false in D_T.*

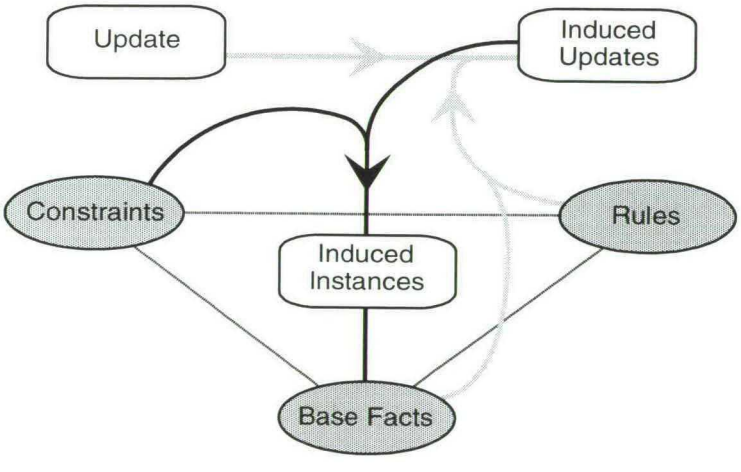


Figure 2.2: An overview of integrity constraint checking based on induced updates

FIGURE 2.2 visualizes PROPOSITION 5. The first step represented by the grey arrows shows the determination of the induced updates by applying the update to the rules with the aid of the fact base. The second step represented by the black arrows shows:

- the determination of the instances of the inconsistency indicators relevant to induced updates, and
- the checking of the instances of the inconsistency indicators with the aid of the fact base.

The next section shows another approach, which is based on potential updates. It prevents to a certain extent the generation of irrelevant induced updates. The next section will elaborate on this method.

2.4.2.2 Integrity Constraint Checking based on Potential Updates

The method based on potential updates follows a strategy that is comparable to the method based on induced updates. It turns out that checking the consistency of the database is done by only evaluating the instantiated inconsistency indicators, which will be called *potential instances*.

DEFINITION 2.25 Let U be an update. Let II be an inconsistency indicator and let L be a potential update relevant to II. The simplified instance of II with respect to L is now called a *potential instance* of II with respect to L.

Note that the update itself can be left out from the definition of potential instances because we know beforehand that the update is always present in the updated database. Checking if an update is in the updated database is redundant. However, a potential update which is not equal to the update itself cannot be left out from a potential instance of a inconsistency indicator, not even when this potential update is ground because it is not certain whether a potential update actually corresponds to an induced update as EXAMPLE 2.7 already showed.

In the method based on potential updates, three phases can be distinguished:

(i) *the generation phase*, in which all potential updates are generated,

(ii) *the selection phase*, in which all relevant potential updates are selected,

(iii) *the evaluation phase*, in which all the potential instances of the inconsistency indicators, which were derived in the previous phase, are evaluated.

When comparing the method based on induced updates to the method based on potential updates, it turns out that the computation of induced updates is postponed to the evaluation phase. Here, in the first phase potential updates are derived instead of induced updates. In the second phase only the relevant ones are selected. After this selection the instantiated inconsistency indicators are determined with respect to these potential updates. When evaluating the generated instantiated indicators, induced updates are derived implicitly. These phases will be explained by using EXAMPLE 2.6.

EXAMPLE 2.13 Suppose we have the situation as in EXAMPLE 2.6. Suppose we also have an inconsistency indicator which expresses that in our database a person cannot be a *parent* and a student at the same time:

$$II_1 : \exists X \exists Y [parent(X, Y), student(X)]$$

Note that the database was consistent before the update, because $student(1)$ does not hold in the database. In EXAMPLE 2.6 the first phase of the method based on potential updates for this specific database is illustrated: given the update $husband(1, 2)$ the potential insertions $husband(1, 2)$, $mother(2, Y)$ and $parent(2, Y)$ are found. For instance, potential update $mother(2, Y)$ is derivable via rule R_1 and the update $husband(1, 2)$. In the second phase, from the potential updates all relevant potential updates are selected, which is $parent(2, Y)$. For all other potential updates it holds that they are not relevant to any inconsistency indicator in the database. In the third phase, first the potential instance of II_1 is derived from the potential update relevant to II_1, i.e., $parent(2, Y)$. This instance is just the simplified instance of II_1 with respect to $parent(2, Y)$, i.e., $parent(2, Y)$, $student(2)$. When evaluating the potential instance of II_1, we see that the potential instance does not hold in the updated database, because of the absence of fact $student(2)$ in the updated database.

Note that by first proving the absence of $student(2)$, $parent(2, Y)$ does not have to be evaluated anymore.

By definition only the relevant potential updates instantiate one or more indicators. PROPOSITION 7 justifies that it is sufficient to check the potential instance of an inconsistency indicator for checking the consistency of the database. The analogue of PROPOSITION 5 with respect to DEFINITION 2.25 is:

PROPOSITION 7 *Let D be a consistent database and let U be an update. Then D_U is consistent iff each potential instance of an inconsistency indicator is false in D_U.*

Proof: The property follows from PROPOSITION 5. For, firstly, all induced instances of inconsistency indicators are instances of some potential instances of inconsistency indicators. Secondly, all potential instances of indicators which are not related to an induced instance of an indicator are false in D_U, because they were already false in the previous database state D. ❏

PROPOSITION 7 can easily be generalized from one update to a transaction.

PROPOSITION 8 *Let D be a consistent database and let T be an update. Then D_T is consistent iff each potential instance of an inconsistency indicator is false in D_T.*

REMARK Note that all relevant induced updates generated by the first method are instantiations of the potential updates generated by the second method. As a consequence, each induced instance of some inconsistency indicator is an instantiation of some potential instance of that inconsistency indicator. The opposite may not hold. It is possible that some potential instance of an inconsistency indicator does not have some corresponding induced instance. Some relevant

potential updates may not have an instance corresponding to some relevant induced update. This follows from the remark after DEFINITION 2.20 and EXAMPLE 2.7, which shows that a potential update may not have any induced update as instance. Note that in EXAMPLE 2.7 we could have made an irrelevant update relevant by specifying an inconsistency indicator to which it is relevant.

Returning to EXAMPLE 2.6, PROPOSITION 7 states that in order to determine whether the database is consistent after the update *husband*(1, 2) or not, it is sufficient to evaluate the potential instance of II_1, i. e. , *parent*(2, *Y*), *student*(2) in the updated database. Because *parent*(2, 10) (and also *parent*(2, 11) but just one evaluation is sufficient) does not hold in the updated database for it is an induced update, and *student*(2) does not hold in the new database state (as in the old database state) the potential instance fails. So, according to PROPOSITION 7 the updated database is consistent. Note that following PROPOSITION 7 has the disadvantage of computing all potential updates, even those for which no inconsistency indicator is relevant for instance, in our example *mother*(2, *Y*).

In general, the advantage of the method based on potential updates, compared to the method based on induced updates, is that we do not spoil any evaluation time for finding instances of irrelevant potential updates. For instance, in the extreme case that for a database no inconsistency indicators were specified, induced updates are derived, which are all irrelevant, while all potential updates are irrelevant and are therefore not evaluated.

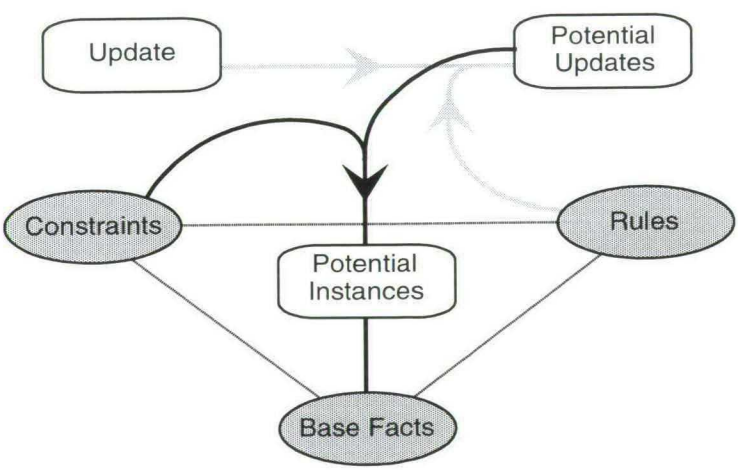


Figure 2.3: An overview of integrity constraint checking based on potential updates

FIGURE 2.3 visualizes PROPOSITION 7. The first step represented by the grey arrows shows the determination of the potential updates by applying the update to the rules without the aid of the fact base. The second step which is represented by the black arrows shows:

- the determination of the instances of the inconsistency indicators relevant to at least one potential update, and
- the checking of the instances of the inconsistency indicators with the aid of the fact base.

As we have seen, proving that an inconsistency indicator is false in deductive databases according to PROPOSITION 7 has a drawback, because some irrelevant potential updates may be generated. In the next chapter, several redundancies in integrity constraint checking in deductive database are studied and classified.

References

[ABW88] KRYSZTOF R. APT, HOWARD A. BLAIR AND ADRIAN WALKER. Towards a Theory of Declarative Knowledge. In J. MINKER, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148, 1988.

[BDM87] FRANÇOIS BRY, HENDRIK DECKER AND RAINER MANTHEY. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In J. W. SCHMIDT, S. CERI AND M. MISSIKOFF, editors, *Advances in Databases Technology, EDBT '88; Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 488–505, Venice, Italy, November 1987. also: ECRC Technical Report KB-16.

[BJ93] PETER BUNEMAN AND SUSHIL JAJODIA, editors. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22-2, Washington, DC, June 1993. ACM Press.

[Bla81] B. T. BLAUSTEIN. *Enforcing Database Assertions: Techniques and Applications*. PhD thesis, Comp. Sc. Dept., Harvard University, Cambridge, Massachusetts, August 1981.

[BM86] FRANÇOIS BRY AND RAINER MANTHEY. Checking Consistency of Database Constraints: a Logical Basis. In Chu et al. [CGO86], pages 13–20.

[BM87] FRANÇOIS BRY AND RAINER MANTHEY. Proving Finite satisfiability of Deductive Databases. In *Proceedings of the Conference Logic and Computer Science*, Lecture Notes in Computer Science. Springer-Verlag, 1987.

[Bro78] M. BRODIE. *Specification and Verification of Database Semantic Integrity*. PhD thesis, University of Toronto, 1978.

[CGO86] WESLEY CHU, GEORGE GARDARIN AND SETSUO OHSUGA, editors. *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, 1986.

[CM89] I-MIN AMY CHEN AND DENNIS MCLEOD. Derived Data Update in Semantic Databases. In PETER M. G. APERS AND GIO WIEDERHOLD, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 225–235, Amsterdam, The Netherlands, 1989.

[Das90] S. K. DAS. *Integrity Constraints in Deductive Databases*. PhD thesis, Dissertation at the Computer Science Department, Heriot-Watt University, Edinburgh, 1990.

[Dec90] HENDRIK DECKER. Drawing Updates from Derivations. In S. ABITEBOUL AND P. C. KANELLAKIS, editors, *Proceedings of the Third International Conference on Database Theory*, volume 470 of *Lecture Notes in Computer Science*, pages 437–451, Paris, France, December 1990.

[Del87] JAMES P. DELGRANDE. Formal Limits on the Automatic Generation and Maintenance of Integrity Constraints. In *Proceedings of the Sixth ACM SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 190–196, March 1987.

[FW83] R. A. FROST AND S. WHITTAKER. A Step Towards the Automatic Maintenance of the Semantic Integrity of Databases. *The Computer Journal*, 26(2):124–133, 1983.

[GA93] P. W. P. J. GREFEN AND P. M. G. APERS. Integrity Control in Relational Databases. *Data & Knowledge Engineering*, 10:187–223, 1993.

[GH86] G. GOOS AND J. HARTMANIS, editors. *International Conference on Database Theory, '86 (ICDT '86)*, volume 243 of *Lecture Notes in Computer Science*, Rome, Italy, 1986. Springer-Verlag.

[IÖ93] NACI S. ISHAKBEYOĞLU AND Z. MERAL ÖZSOYOĞLU. On the Maintenance of Implication Integrity Constraints. In VLADIMÍR MAŘÍK, JIŘÍ LAŽANSKÝ, AND ROLAND R. WAGNER, editors, *Lecture Notes in Computer Science*, volume 720, pages 221–232, Prague, Czech Republic, September 1993. Springer-Verlag.

[Ker86] LARRY KERSCHBERG, editor. *Expert Database Systems: Proceedings from the First International Workshop*. Charleston, Sc., 1986.

[Kob85] ISAMU KOBAYASHI. *An Overview of Database Management Technology*, chapter 2, pages 49–219. Plenum Pres, New York-London, 1985.

[Kun85] C. KUNG. A Tableaux Approach for Consistency Checking. In A. SERNADAS, JR. J. BUBENKO AND A. OLIVÉ, editors, *Information Systems: Theoretical and Formal Aspects*, pages 191–207, North-Holland, 1985. IFIP, Elsevier Science Publishers B.V.

[Kwa91] KAREN L. KWAST. A Deontic Operator for Database Integrity. In J.-J. MEIJER AND R. WIELINGA, editors, *Proceedings of DEON'91; International Workshop on Deontic Logic in Computer Science*, pages 263–279, Amsterdam, 1991.

[LR84] TOK-WANG LING AND P. RAJAGOPALAN. A Method to eliminate Avoidable Checking of Integrity Constraints. In *Trends and Applications*, pages 60–68. IEEE, Computer Society Press, 1984.

[MWW89] J.-J. MEYER, H. WEIGAND AND R. WIERINGA. Specification Language for Static, Dynamic and Deontic Integrity Constraints. In *Proceedings of the Second Symposium on Mathematical Fundamentals of Database Systems*, pages 347–366, Visegrad, Hungary, June 1989.

[NA86] H. NOBLE AND T. ABBOD, editors. *Proceedings of the Fifth British National Conference on Databases, BNCOD 5*, Canterbury, UK, 1986.

[Nic79] J. M. NICOLAS. A Property of Logical Formulas corresponding to Integrity Constraints on Data Base Relations. In *Proceedings of the Workshop on Formal Bases for Data Bases*, Toulouse, 1979.

[Nic82] J. M. NICOLAS. Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, 18(3):227–253, 1982.

[Ple93] DIMITRIS PLEXOUSAKIS. Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases. In RAKESH AGRAWAL, SEÁN BAKER AND DAVID BELL, editors, *Proceedings of the Nineteenth Conference on Very Large Data Bases*, pages 146–157, Dublin, Ireland, 1993.

[Qia88a] XIAOLEI QIAN. An Effective Method for Integrity Constraint Simplification. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 338–345, Los Angeles, California, USA, February 1988.

[Qia88b] XIAOLEI QIAN. Reasoning about Constraints and Updates in Relational Databases. Technical report, Stanford University, 1988.

[QS87] XIAOLEI QIAN AND DOUGLAS R. SMITH. Integrity Constraint Reformulation for Efficient Validation. In PETER M. STOCKER AND WILLIAM KENT, editors, *Proceedings of the Thirteenth Conference on Very Large Data Bases*, pages 417–425, Brighton, England, 1987.

[QW86] XIAOLEI QIAN AND GIO WIEDERHOLD. Knowledge-based Integrity Constraint Validation. In Chu et al. [CGO86], pages 3–12.

[RM88] K. V. S. V. N RAJU AND A. K. MAJUMDAR. Fuzzy Functional Dependencies and Lossless Join Decomposition of Fuzzy Relational Database Systems. *ACM Transactions on Database Systems*, 13(2):129–166, June 1988.

[SK88] NORBERT SÜDKAMP AND PETER KANDZIA. Enforcement of Integrity Constraints in a Semantic Data Model. In E. BÖRGER, H. KLEINE BÜNING AND M. M. RICHTER, editors, *First Workshop on Computer Science Logic, CSL'87*, volume

385 of *Lecture Notes in Computer Science*, pages 313–328, Duisburg, FRG, 1988. Springer-Verlag.

[Sto75] M. STONEBRAKER. Implementation of Integrity Constraints and views by Query Modification. In *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data*, pages 65–78, San Jose, CA, June 1975.

[SV85] ERIC SIMON AND PATRICK VALDURIEZ. Design and Analysis of a Relational Integrity Subsystem. Technical Report DB-015-87, MCC, Austin, USA, 1985.

[SW94a] RICHARD T. SNODGRASS AND MARIANNE WINSLETT, editors. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23-2, Minneapolis, Minnesota, May 1994. ACM Press.

[SW94b] S. M. SRIPADA AND B. WÜTHRICH. Cumulative Updates. In JORGE BOCCA, MATTHIAS JARKE AND CARLO ZANIOLO, editors, *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 534–545, Santiago, Chile, September 1994.

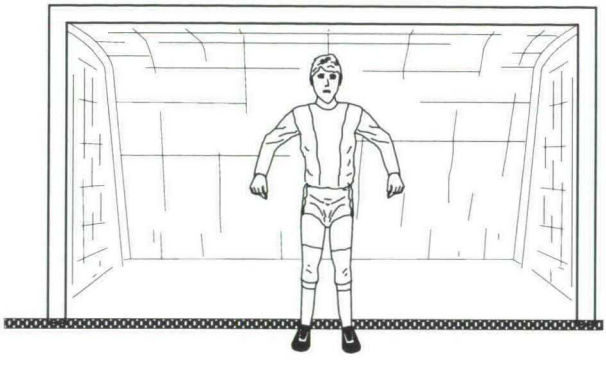
[Ull88] JEFFREY D. ULLMAN. *Principles of Database and Knowledge Base Systems*, volume I. Computer Science Press, Stanford University, 1988.

[WMW89] R. WIERINGA, J.-J. MEYER AND H. WEIGAND. Specifying Dynamic and Deontic Integrity Constraints. *Data & Knowledge Engineering*, 4(2):157–189, 1989.

[WSK83] WOLFGANG WEBER, WOLFFRIED STUCKY AND JAKOB KARSZT. Integrity Checking in Data Base Systems. *Information Systems*, 8(2):125–136, 1983.

[WY92] KE WANG AND LI YAN YUAN. Preservation of Integrity Constraints in Definite DATALOG Programs. *Information Processing Letters*, 44(4):185–193, December 1992.

"... but some integrity constraints were specified ..."



Chapter 3

Redundancies in Integrity Constraint Checking

This chapter will show some redundancies that may appear in methods for checking the integrity constraints of deductive databases. Note that the previous chapter showed the main assumption for checking the consistency of deductive databases, i. e. , the database is supposed to be consistent before any update. This assumption alleviates the need for full checks of all constraints. However, when restricting ourselves to only relevant instantiated inconsistency indicators, our check can still contain a lot of redundancy.

Note that fact base access on secondary storage is relatively expensive. Therefore, each redundant fact base access must be avoided. In this chapter, several types of redundancy are distinguished in the consistency check of a deductive database after an update. Here, these types are described without looking at a specific integrity checking method. The redundancy types described in this chapter are used for the comparison of several methods for checking the integrity constraints in deductive databases, which are described further on in this thesis. A redundancy type may be typical for some method, while other types do not appear in that method. The concepts of induced update and potential update, which were defined in the previous chapter, are used for describing and classifying these redundancies. In this classification of redundancies three phases in the integrity constraint check are distinguished, i. e. , *the generation phase*, *the selection phase* and *the evaluation phase*, which are recognized in the methods based on induced and potential updates, as was shown in the previous chapter. In the following, when in a context induced updates or potential updates are involved, while the used type of updates are of no concern in that context, induced and potential updates are called *derived updates*.

3.1 Redundancy by Duplicates

In a consistency check, based on some method for checking the integrity of the database, certain parts of the evaluation may have been done repeatedly. This duplication of work is not necessary and has to be avoided. In this section, the possible causes of these redundancies are described. Duplicates can appear in each of the phases of an integrity constraint checking process, e. g. , in the generation phase, in the selection phase or in the evaluation phase of the method of induced updates resp. potential updates.

3.1.1 Redundancy by Duplicates in Transactions

The most evident kind of redundancy is the appearance of duplicates in a transaction. As a result all the induced updates of a duplicate update in the transaction are also duplicate induced updates. When a relevant inconsistency indicator has to be evaluated for such a duplicate induced update in the transaction, it is evident that this inconsistency indicator has been evaluated before.

So, duplicate updates in the transaction must be avoided. Duplicate induced updates are not only generated by duplicates in the transaction, as the next section shows.

3.1.2 Redundancy by Duplicates among Derived Updates

When considering some method the intermediate results of a check can contain duplicates. For instance, in the methods based on induced or potential updates, redundant evaluations of induced resp. potential instances of indicators may appear, when some duplicate derived updates are generated in the generation phase of each method. The next example illustrates this kind of duplication.

EXAMPLE 3.1 Let D be a deductive database with the following rule and fact base:

RULES

R_1: $mother(X, Y) \longleftarrow husband(Z, X), father(Z, Y)$

R_2: $parent(X, Y) \longleftarrow father(X, Y)$

R_3: $parent(X, Y) \longleftarrow mother(X, Y)$

R_4: $is\_child(Y) \longleftarrow parent(X, Y), age(Y, N), N < 15$

FACT

F_1: $husband(1, 2)$

F_2: $age(10, 5)$

INCONSISTENCY INDICATOR

II_1: $\exists Y[is\_child(Y), heavy\_job(Y)]$

The indicator prohibits that a child has a heavy job and that a heavy job is assigned to a child. When this database is updated by the insertion $father(1, 10)$, then two induced updates with respect to relation *parent*, i. e. , $parent(1, 10)$ and $parent(2, 10)$, are derived. Each of these induced updates leads to the induced update $is\_child(10)$. When using each of these induced updates in the remainder of the method, each appearance of $is\_child(10)$ would lead to an evaluation of an update instance of II_1, namely, $heavy\_job(10)$.

In a breadth first computation of induced updates this redundant checking of the same update instance is avoided, because all redundant induced updates are known before applying them to inconsistency indicators. In a depth first computation of induced updates, we should keep a list of the induced updates, which already have been handled in the consistency check, skipping each

next induced update that is already present in the list. Note that in the second evaluation technique induced updates, which are induced by some duplicate induced update, do not have to be computed anymore, because they are duplicates as well. Note that the effect of duplicate derived updates could lead to duplicate instances of inconsistency indicators. However, this is not always caused by duplicate derived updates as the next section shows.

3.1.3 Redundancy by Duplicate Instances of an Indicator

Duplicate instances of inconsistency indicators are not always the result of the generation of duplicate induced or potential updates. Duplicate instances of inconsistency indicators may also occur, when several different induced updates exists, which all are relevant to the same indicator and they instantiate the inconsistency indicator in the same way, i.e., the literals of the inconsistency indicator are identically instantiated for these induced updates resulting in checks that are essentially the same. EXAMPLE 2.12 already showed a variant of this kind of redundancy. In that example two induced updates *parent*(2, 10) and *parent*(2, 11) were derived leading to the induced instances of the inconsistency indicator *parent*(X, Y), *student*(X), i.e., *student*(2) for both induced updates. So, *student*(2) does not have to be evaluated a second time. A duplicate instantiation of an inconsistency indicator is not necessarily caused by comparable updates in the same relations. Duplicate instances of inconsistency indicators may also be the result of updates in several relations. This is illustrated by the following example.

EXAMPLE 3.2 Let D be the deductive database consisting of the following rule base and inconsistency indicator:

RULES

R_1: *mother*$(X, Y) \longleftarrow$ *husband*(Z, X), *father*(Z, Y)

R_2: *parent*$(X, Y) \longleftarrow$ *father*(X, Y)

R_3: *parent*$(X, Y) \longleftarrow$ *mother*(X, Y)

INCONSISTENCY INDICATOR

II_1: $\exists X \exists Y$[*parent*(X, Y), *student*(X)]

Suppose the transaction consists of insertions *father*(1, 10) and *student*(1). The two instantiations of II_1 are found, namely *parent*(1, 10), *student*(1) and *parent*$(1, Y)$, *student*(1). Note that the latter instantiation subsumes the first one; so, the first one should not be evaluated, when the second one is evaluated first.

Note that to avoid duplicate evaluations of inconsistency indicators a solution analogue to that for duplicate derived updates is thinkable. In a breadth first computation of update instances, first all update instances of the inconsistency indicators are computed. Thereafter, all duplicates and subsumed ones are removed and, when needed, further compiled. In the depth first computation all evaluated update instances are listed and a new update instance is only evaluated, when it is not already present in this list.

Not only instantiations of a particular inconsistency indicator can lead to duplicate evaluations of instances of that inconsistency indicator, but different inconsistency indicators can also lead to duplicate evaluations of (parts of) these inconsistency indicators.

3.1.4 Redundancy by Duplicate Side Literals of Different Indicators

When instantiating inconsistency indicators, the resulting instantiated inconsistency indicators may be identical, as we have seen in the section on redundancy by duplicate instances of an indicator. However, identical instances of indicators may be caused by instantiations of several indicators instead of one particular indicator. The next example shows this kind of redundancy.

EXAMPLE 3.3 Let D be a database with the following rule, fact and inconsistency indicator base:

RULES

R_1: $mother(X, Y) \longleftarrow husband(Z, X), father(Z, Y)$

R_2: $parent(X, Y) \longleftarrow father(X, Y)$

R_3: $parent(X, Y) \longleftarrow mother(X, Y)$

R_4: $dependent(Y, X) \longleftarrow mother(X, Y), employed(X), student(Y)$

FACTS

F_1: $employed(2)$

F_2: $student(10)$

F_3: $husband(1, 2)$

INCONSISTENCY INDICATORS

II_1: $\exists X \exists Y[parent(X, Y), student(X)]$

II_2: $\exists X \exists Y[dependent(Y, X), student(X)]$

Let $father(1, 10)$ be an update to this database. Besides $mother(2, 10)$, which is an irrelevant induced insertion, other relevant induced insertions are derived as well.
First, the induced insertions $parent(1, 10)$ and $parent(2, 10)$ are both relevant to II_1. Second, the induced insertion $dependent(10, 2)$ is relevant to II_2. The induced instances of II_1 with respect to $parent(1, 10)$ and $parent(2, 10)$ respectively are both equal to the induced instance of II_2 with respect to $dependent(10, 2)$, namely $student(2)$.

This type of redundancy may not only occur for one particular update but may also occur for several updates in a transaction.

Besides exact duplicates, some derived update instance of an indicator may be subsumed by another and can therefore be skipped in the checking process. For instance, suppose two derived

updates $a(X, c, d)$ and $a(X, c, Y)$ are derived by applying some integrity checking method. The derived update $a(X, c, d)$ is a special case of derived update $a(X, c, Y)$ and is therefore redundant in the remainder of the method. We can consider this so called *redundancy by subsumption* as a general case of redundancy by duplicates and is handled in the same way. Redundancy by the appearance of duplicates is not the most interesting type of redundancy. This type of redundancy can easily be avoided by keeping track of intermediate results in the integrity checking process. However, when ignoring this kind of redundancy, the redundancy may be enormous.

Other more hidden types of redundancy are studied in the remainder of this chapter. When considering these redundancies for integrity constraint checking methods in general, we assume that such methods handle redundancy by duplicates well.

3.2 Redundancy in the Generation of Intermediate Results

During the consistency check several intermediate results may be derived. Some of these intermediate results do not really contribute to the necessary check. However, determining the minimal set of facts stored in the database that are needed to complete the check, may never be reachable or even may not be desirable as is pointed out in this section. In this section, the effectiveness and relevance of intermediate results are studied.

3.2.1 Redundancy by Ineffective Intermediate Results

The definition of ineffective induced updates reflects precisely the kind of redundancy that is handled in this section. Although this type of redundancy could be seen as a variant of redundancy by duplicates, because a duplicate is now present in the old database state, it is studied in a separate section. The reason for handling this kind of redundancy becomes clear when reading this section. Suppose a deductive database is updated by a transaction consisting of one update. An ineffective induced update is present in the old database state. So, the relevant inconsistency indicators with respect to the ineffective induced update, were already fulfilled in the previous database state and do not have to be checked again.

Preventing redundancies by *ineffective updates* can lead to a loss of efficiency. For instance, in determining whether an induced update is effective or not, one has to evaluate this update. When it turns out to be ineffective, the evaluation of the related inconsistency indicators is redundant. However, when it is effective, then the evaluation of the update was a wasted effort. So, in order to decide if the execution of such a test for effectiveness, which is called the *effectiveness test*, makes sense, each time an induced update is derived one must have some knowledge about

(i) the expected percentage of induced updates that are ineffective,

(ii) the expected time needed for the ineffectiveness test,

(iii) the expected time needed to evaluate an inconsistency indicator.

Because the time needed for any evaluation is strongly related to the number of database accesses needed in the evaluation, the decision of using such a test concerns the content of the database. The number of facts, rules and inconsistency indicators as well as the interaction between those components of the database and updates are influencing the values of these three items. The database management system must keep statistics of the timings of checks with and without the effectiveness test for various updates, in order to decide for which kind of updates the effectiveness test may lead to a more efficient evaluation. Therefore, it may not always be wise to try to minimize this kind of redundancy at all cost.

3.2.2 Redundancy by Irrelevant Intermediate Results

In the computation of all relevant induced updates, irrelevant intermediate results may be derived, i. e. , results that do not participate in the minimal check that is possible. For example, in EXAMPLE 2.12 some irrelevant induced updates were generated in order to derive the relevant induced updates. When returning to EXAMPLE 2.12, we see that any computation of the *mother*-relation is an example of this kind of redundancy. FIGURE 3.1 shows the computation of all *mother*-facts that are induced by the update *husband*(1, 2), when the database contains the facts *father*(1, 10), *father*(1, 11) and *father*(1, 12). Each of these *mother*-facts is redundant because there is no inconsistency indicator that is directly related to the *mother*-relation. The remark after EXAMPLE 2.12

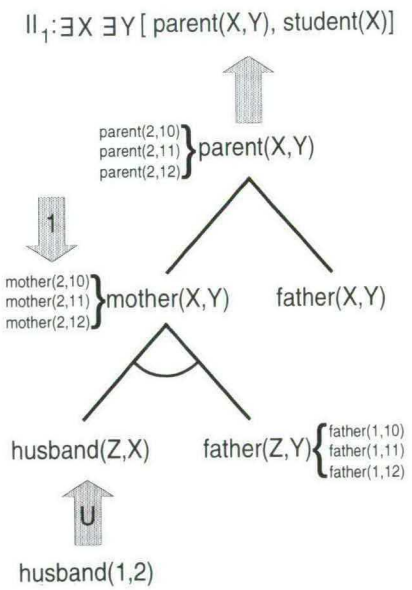


Figure 3.1: Redundancy by irrelevant induced updates

states that these irrelevant induced updates were indispensible for the method based on induced updates for deriving the relevant induced updates, for instance *parent*(2, 10), *parent*(2, 11) and *parent*(2, 12) in the situation of FIGURE 3.1. Generally, this is not always the case, as EXAMPLE 2.13 showed.

REMARK In the induced update method as well as in the potential update method the irrelevant induced updates were accessed, when the induced resp. potential instance of the inconsistency indicator was evaluated. However, cases may exist in which irrelevant induced updates can be avoided, as is stated in the note after EXAMPLE 2.13. In the potential update method, irrelevant potential updates may have been redundantly generated. However, this redundancy is less influencial than a redundancy by generating redundant induced updates, because in generating potential updates the fact base will not be accessed, which is rather inexpensive in comparison to an induced update, for which the database has to be accessed.

Redundancy by irrelevant intermediate results can appear in a more subtile way, when an inconsistency indicator that has to be checked is instantiated. For example, suppose that in FIGURE 3.1 instead of the full inconsistency indicator:

$$II_1 : \exists X \exists Y [parent(X, Y), student(X)]$$

the specified inconsistency indicator is:

$$II_1' : \exists Y [parent(1, Y), student(1)]$$

which expresses that in this database, for some reason, only person 1 cannot be a student and a parent at the same time. Then compared to the previous situation *parent*(2, 10), *parent*(2, 11) and *parent*(2, 12) are no longer relevant.

This example shows that this type of redundancy is part of the redundancy in the generation of intermediate results, because in deriving the relevant inconsistency indicators even more redundant intermediate results may be derived compared to the case of uninstantiated inconsistency indicators. In other words, in the case of an uninstantiated inconsistency indicator some relevant derived updates may be irrelevant in the case the inconsistency indicator is instantiated.

3.2.3 Redundancy by Intermediate Results

The redundancies described in the previous sections, which are related to ineffective and irrelevant derived updates, are presented in a unifying context. This is done along the following example.

EXAMPLE 3.4 Let D be a database with the following rule, fact and inconsistency indicator base:

RULES

R_1: $mother(X, Y) \longleftarrow husband(Z, X), father(Z, Y)$

R_2: $parent(X, Y) \longleftarrow father(X, Y)$

R_3: $parent(X, Y) \longleftarrow mother(X, Y)$

R_4: $mother(X, Y) \longleftarrow child(Y, X)$

FACTS

F_1: $father(1, 10)$

F_2: $father(1, 11)$

F_3: $child(10, 2)$

F_4: $student(3)$

INCONSISTENCY INDICATOR

II_1: $\exists X \exists Y[parent(X, Y), student(X)]$

Note that $mother(2, 10)$, $parent(2, 10)$, $parent(1, 10)$ and $parent(1, 11)$ are derivable in D and that D is consistent with respect to II_1. Let $U : husband(1, 2)$ be an insertion to D. As a result several induced updates in the relations of *mother* and *parent* are derived. These induced updates are $mother(2, 10)$, $mother(2, 11)$, $parent(2, 10)$ and $parent(2, 11)$. Note that $mother(2, 10)$ and $mother(2, 11)$ are both irrelevant, while even $mother(2, 10)$ is ineffective. On the other hand, $parent(2, 10)$ and $parent(2, 11)$ are both relevant, of which only $parent(2, 11)$ is effective. When looking at the potential updates derived from update U, we see that $mother(2, Y)$ and $parent(2, Y)$ are derived, of which $mother(2, Y)$ is irrelevant but effective and $parent(2, Y)$ is relevant and effective. This is shown in the pyramid of FIGURE 3.2.

In this figure, triangles are important for its interpretation. The semantics of this pyramid is analysed in the following way. The whole pyramid consists of derived updates that result from U. In fact, the pyramid consists of potential updates represented by the triangle of which the basis is marked with *potential updates*. As we saw in 2.4.2.2, some of them can be instantiated to induced updates. These induced updates are represented by the triangle of which the basis is marked with *induced updates*. However, the potential updates may have other instances, which do not correspond to induced updates. These instances are present in the white area of the pyramid. Now, the derived updates are observed from two other perspectives. On the left side of the pyramid a distinction is made between effective and ineffective derived updates. The effective part of all derived updates is enclosed in the triangle of which the side is marked by *effective*. In the other part of the pyramid all inefffective derived updates are represented. On the right side of the pyramid a distinction is made between relevant and irrelevant derived updates. The relevant part of the derived updates is enclosed in the triangle of which the side is marked by *relevant*. In the other part of the pyramid all irrelevant derived updates are represented. Note that the inner core of the pyramid is the most interesting part with respect to the inconsistency indicator. This part, represented by the smallest triangle in the figure, actually influences the consistency of the database, as the main assumption of the previous chapter implies.

REMARK The concepts of relevant and effective derived update are *orthogonal*. Relevant derived updates as well as irrelevant derived updates may be either effective or ineffective. So, the

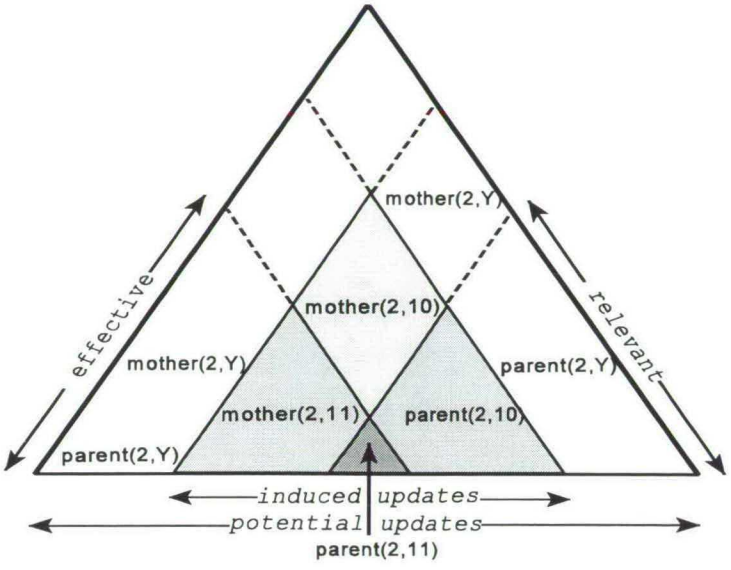


Figure 3.2: Redundancy of the first type

converse holds too: effective derived updates as well as ineffective derived updates may be either relevant or irrelevant, also holds. Note, that instances of some effective potential update may correspond to effective as well as ineffective induced updates and that instances of a some relevant potential update may correspond to relevant as well as irrelevant induced updates. This last remark is illustrated, when in the example II_1 is replaced by the indicator $\exists X[parent(X, 11), student(X)]$. In this case $parent(2, 10)$ becomes irrelevant, while $parent(2, 11)$ still is relevant.

We call the redundancy, consisting of computing all irrelevant and ineffective intermediate results, a *redundancy of the first type*.

3.3 Redundancy in the Selection of Inconsistency Indicators

Sometimes in the selection phase of a method, i. e. , the phase in which the inconsistency indicators are chosen that have to be evaluated, a selection of an inconsistency indicator is not necessary. For instance, in a naive method where all inconsistency indicators have to be evaluated the selection of irrelevant inconsistency indicators is an example of this kind of redundancy. However, the redundant selection may have a more hidden cause. A redundant selection leading to a redundant evaluation of a partially instantiated inconsistency indicator may be caused by the fact that the instantiated inconsistency indicator contains a partial relation which is empty, i. e. ,

no instantiation of this relation holds in the updated database. For instance, this can happen in the method based on potential updates. Although in EXAMPLE 2.6 the update *husband*(1, 2) can influence the inconsistency indicator II_1, the extensional database in this case does not have to imply *mother*-facts and therefore also no new *parent*-facts. If the *father*-relation does not contain facts for person 1, then no induced updates with respect to the *mother*-relation are found. So, there are no new *parent*-facts derivable in the database. This means that the evaluation of II_1 is a redundant evaluation, because the evaluation of *parent*(X, Y) in II_1 does not involve *parent*-facts, which were not present before the update. This situation is illustrated in FIGURE 3.3, where the dotted lines show the part of the database that does not change.

Note that the occurrence of this type of redundancy is highly dependent on the particular database state and therefore can only be determined at run-time. Therefore, during an inconsistency check it is important to find out with a minimum of database accesses if this situation occurs. This kind

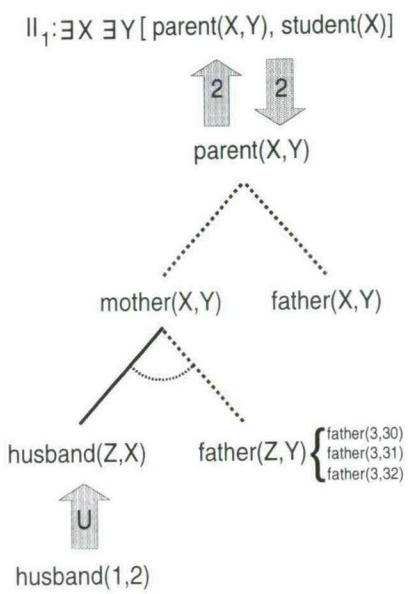


Figure 3.3: Redundancy of the second type

of redundancy does not appear in the method based on induced updates, because by generating the induced updates first, one can pick out those inconsistency indicators that are really influenced by the update. This is a serious problem in the method based on potential updates in which a relevant inconsistency indicator turns out to be irrelevant when actually exploring the database.

We call this kind of redundancy a *redundancy of the second type*.

3.4 Redundancy in the Evaluation of Inconsistency Indicators

The third type of redundancy is the redundancy that appears in the evaluation of the selected inconsistency indicators. It involves an evaluation of parts of the database that are not affected by the update. For instance, in the method based on potential updates the update *husband*$(1, 2)$ may cause an implicit update in the *parent*-relation because of a change in the *mother*-relation. The

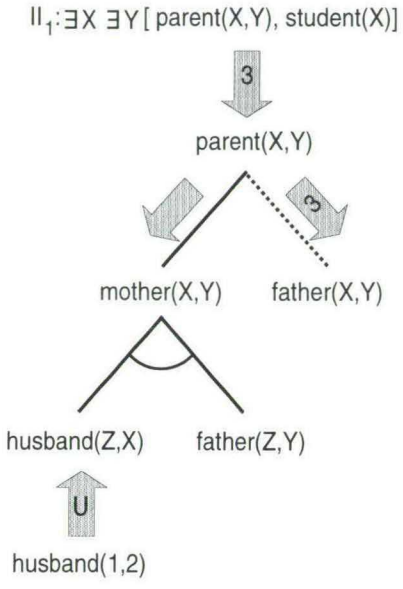


Figure 3.4: Redundancy of the third type

resulting evaluation of *parent*$(2, Y)$, *student*(2) will lead to a search for a change of the *mother*-relation as well as the *father*-relation. But the *father*-relation did not change by the update. Therefore, the evaluation of the indicator by going into the right branch of our tree is redundant. This situation is shown in FIGURE 3.4.

Besides a check of an updated branch of such a tree, this could lead to a check of branches which are unchanged. We call this kind of redundancy a *redundancy of the third type*.

REMARK Redundancy of the third kind may exist also in the method based on induced updates. For instance, the update *husband*$(1, 2)$ could lead to an induced update in the *parent*-relation, resulting in an evaluation of an induced instance of II_1. This evaluation will search through the *father* definition part of the *parent*-relation, which is clearly not changed.

Redundancy of the third type can lead to an enormous overhead in case of dependency trees, representing the intensional database, which are deeply and widely branched. This is represented in FIGURE 3.5. In this figure the dependency tree is represented by an and/or-tree, in which and-nodes are connected by an arc in the branches to the and-nodes. The parent node of such and-nodes represents a head of a rule and the and-nodes represent the body of that rule. The continuous lines in FIGURE 3.5 show the influence of the update, i. e. , the relations that are updated by the update. The dotted lines show the part of the database that does not change. But evaluating the expression in the top node means that all the branches will be searched for a change, even the dotted branches. Note further that in case of a combination of redundancy of the second and third type, the overhead can become extremely large, as the second picture in FIGURE 3.5 shows.

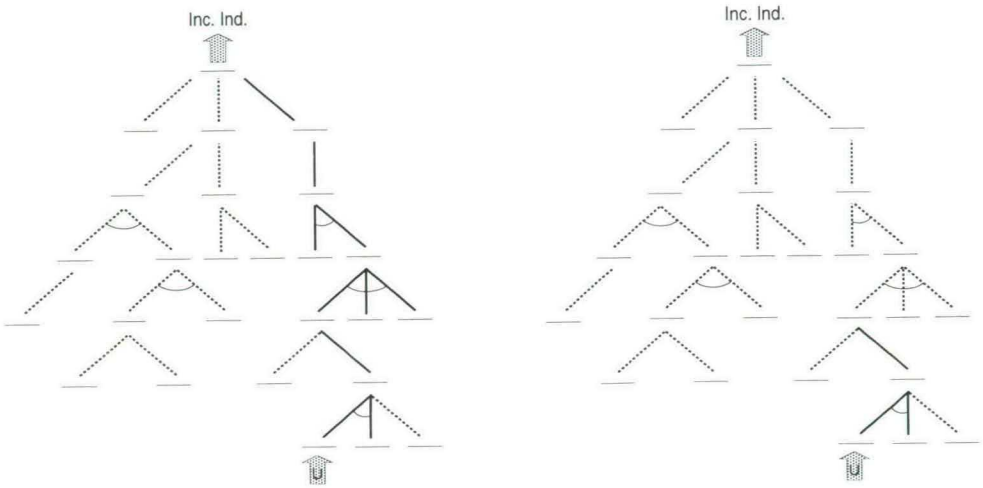


Figure 3.5: Redundancy of the third type in case of large dependency trees

3.5 Redundancy by Neglecting the Relation between Updates

The updates in a transaction are often in some way related to each other. When they are not related, there may be no reason for them to be in the same transaction and they could be presented to the database as independent updates. However, when there is some link between updates in the transaction, knowledge about such a link can be of great use in finding inconsistency indicators that are more likely to succeed than others. Redundantly evaluating inconsistency indicators which fail, while there exists an inconsistency indicator that succeeds, must be avoided. It may seem that one cannot predict, whether an inconsistency indicator succeeds or not. However, knowledge about the statistically optimal order of evaluation of inconsistency indicators can in the long run lead to a significant gain of efficiency. We illustrate this by a simple example.

EXAMPLE 3.5 Let D be a database, which contains the following inconsistency indicators:

II_1: $\exists X[p(X), r(X)]$

II_2: $\exists X \exists Y[q(X), s(X, Y)]$

II_3: $\exists X[p(X), q(X)]$

In D p and q are base relations and r and s are derived relations. Let $T = \{p(a), q(a)\}$ be a transaction. Suppose that $r(a)$ and $s(a, Y)$ do not hold in D_T. When the order in which the indicators are presented here is used in the consistency check, this results in the evaluation of $r(a)$ and $s(a, Y)$ first, before the evaluation of II_3. It is obvious that II_3, in which p and q appear, should be checked first, because it can be completely evaluated without any database access. So, first look if there is some indicator that can be evaluated by only using the transaction.

Note that this kind of redundancy is only relevant if at least one of the inconsistency indicators will succeed in the updated database. When they all fail, each of the inconsistency indicators is checked. So, in this case the order of evaluation of the indicators does not matter.

There is another argument for evaluating II_3 first. It contains base relations only. Evaluating a derived relation could cause a tremendous database search, when the relations are defined in terms of many other relations. This could lead to an explosion of database accesses, which is prevented when II_3 succeeds. A solution for these problems is to compute all base relations that are involved when evaluating an inconsistency indicator. So, when this knowledge is linked to each inconsistency indicator, a measure for evaluating some inconsistency indicators before others could be the percentage of base facts, appearing in the transaction, in the set of base facts that are involved in evaluating the inconsistency indicator. Indicators that consist only of base relations, which also appear in the transaction, must have the highest priority.

This section showed that the order of inconsistency indicators can play an important role in the gain of efficiency. However, the choice of the evaluation order of inconsistency indicators can vary from database to database.

We call this kind of redundancy a *redundancy of the fourth type*.

3.6 Redundancy by Replacement

Replacements are often seen as a deletion followed by an insertion, which is in general not the best way to look at replacements. From the integrity checking point of view, this can lead to a lot of redundancy. In the literature, replacement with respect to integrity constraint checking seems to be overlooked, but it is rather essential. The next example illustrates redundancy if a replacement is considered as an insertion after a deletion.

EXAMPLE 3.6 Let D be a database containing *employee*-facts of the following form:

employee(Emp_id, $Address$, $Function$, $Date\_of\_Birth$, $Date\_of\_Retirement$, $Mngr\_id$)

and for which the inconsistency indicators

II_1: $employee(E, A_1, F_1, B_1, R_1, M), employee(M, A_2, F_2, B_2, R_2, E)$

II_2: $employee(E, A, assistent, B, R, M), E > M$

II_3: $employee(E, A, manager, B, R, M), retirement\_age(B, R, N), N < 60$

are specified, where each variable in the inconsistency indicators is existentially quantified and *retirement_age* is an evaluable predicate, which has the date of birth and the date of retirement as input arguments and the computed age at the date of retirement as output. The inconsistency indicators express that a state in which an employee has a manager for whom he himself is a manager, or when the employee is an assistent with an identifier greater than that of its manager, or when the employee is a manager who retires before the age of 60, is prohibited.
Let F : $employee(12, Kings\ Road\ 15\ London, assistent, 12/04/62, 01/07/96, 92)$ be a fact in the database. The represented employee is transferred to another department of the company in Paris, which leads to the replacement of this fact by

$$F' : employee(12, Rue\ d'Amerique\ 66\ Paris, assistent, 12/04/62, 01/07/96, 92).$$

If we consider this replacement as a deletion of F followed by the insertion of F', all inconsistency indicators are checked again. However, when looking at the replacement more carefully, only the address is replaced, while none of the inconsistency indicators constrains the address in any way. In other words, evaluation of the inconsistency indicators for F or F' leads to the same update instances of the inconsistency indicator; so, obviously with respect to the consistency of the database, nothing has changed. Therefore, each inconsistency indicator, which is not influenced by a replacement, is redundantly evaluated.

Redundancy by replacement is in fact a redundancy that appears in the selection phase. When not handled properly, a method selects inconsistency indicators in which the relations did not change. In the literature, up till now integrity checking methods for deductive databases do not cope with this kind of redundancy; however, in the method presented in this thesis this kind of redundancy can easily be avoided.

3.7 Related Research

Redundancies in the evaluation of inconsistency indicators, may be caused by the order of their subgoals and is the concern of query optimization techniques. This is not the kind of redundancy that is studied here. Although, in the literature, a lot of attention is paid to redundancy in query evaluation, which can be found in papers devoted to query optimization, there is not done much fundamental research to the causes of redundancies in integrity checking.

However, in [LL94] Lee and Ling explicitly describe a type of redundancy, which is caused by ignoring the information that is present in the constraint itself, as in the situation when an inconsistency indicator is instantiated. As we have seen in the case of instantiated inconsistency

indicators, ignoring the information that is present in the indicator could lead to an unnecessary computation of induced updates. Lee and Ling discovered a more general redundancy caused by evaluable predicates that appear in inconsistency indicators. Their idea is to evaluate these predicates as soon as possible in order to prevent database accesses caused by evaluation of other predicates in indicators. For instance, let

$$II_1 : \exists X \exists Y[parent(X, Y), student(X), eval(X)]$$

be an indicator that contains some evaluable predicate *eval*, which only depends on variable X.

Lee and Ling show that their optimization technique can be added to integrity checking methods in order to avoid this kind of redundancy. For instance, in the method of potential updates, first we could select the potential updates that are relevant to the indicator. When a potential update with respect to relation *parent* or *student* exists, which binds X in the indicator, then we should first evaluate *eval* before finding all induced updates, that correspond to the potential updates, in order to save database access time when $eval(X)$ does not hold. In fact when X is bound the inconsistency indicator is interpreted as:

if $eval(X)$, then evaluate $\exists Y[parent(X, Y), student(X)]$.

When X is not bound, the inconsistency indicator is evaluated in the regular way. Note that an inconsistency indicator which is instantiated by itself can be seen as an indicator with some evalable predicate, e. g. , $\exists Y[parent(1, Y), student(1)]$ can be expressed as $\exists Y[parent(X, Y), student(X), X = 1]$. However, this type of redundancy is not a new type. Each redundant database access, caused by this kind of redundancy, can be caused either by a redundant generation of irrelevant intermediate results or by a redundant selection of an inconsistency indicator.

In [WSK83] another redundancy concerning integrity checking in the case of aggregate constraints is studied. Instead of checking a full aggregate constraint after an update, which could lead to accessing all base facts that were needed to derive the value related to the aggregate function again, while we only need the previous aggregate value together with the values of the updates that contribute to this value. For instance, when an aggregate constraint I states that a relation R must not exceed the number of n tuples, and a tuple from R is removed, it is awkward to count the number of tuples in R again. Instead of this recount, together with I the number of tuples is stored. Say this number is v. So, checking I corresponds to find out the number of tuples added to this relation, say m resulting from the update and compute $v + m$ to see if it does not exceed n, as stated by I. Hence, this constraint can be checked without accessing the database.

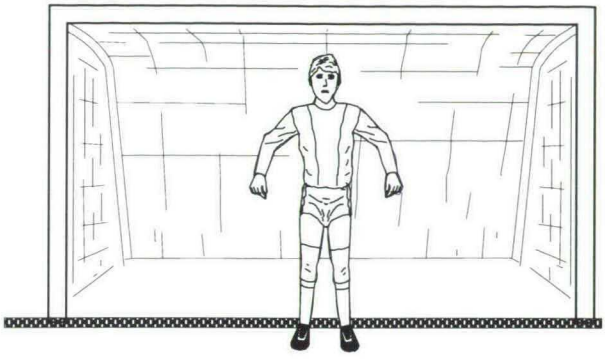
References

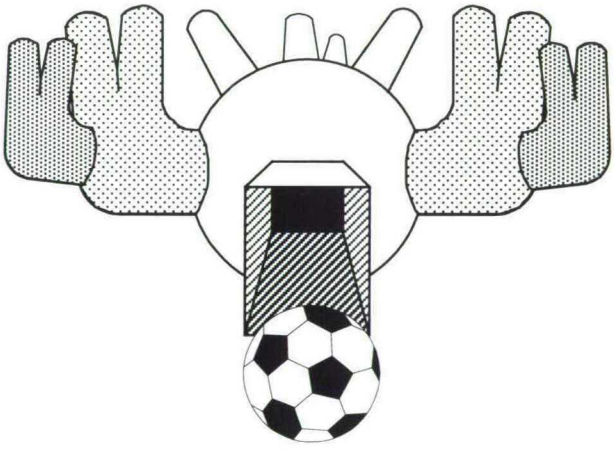
[LL94] Sin Yeung Lee and Tok Wang Ling. Improving Integrity Constraint Checking for Stratified Deductive Databases. In Dimitris Karagiannis, editor, *Lecture Notes*

in Computer Science, volume 856, pages 591–600, Athens, Greece, September 1994. Springer-Verlag.

[WSK83] WOLFGANG WEBER, WOLFFRIED STUCKY AND JAKOB KARSZT. Integrity Checking in Data Base Systems. *Information Systems*, 8(2):125–136, 1983.

"... and the database was deductive ..."





Chapter 4

FICCS: Fact Integrity Constraint Checking System

In deductive databases it is important that the information we obtain from the database is consistent with our view of the world. However, there may be several causes for deriving inconsistent information from the system. In 2.2 we noted that an inconsistent set of integrity constraints could lead to problems. Contradictory information may be derived from the database when the set of integrity constraints is inconsistent. Some examples of inconsistencies or redundancies in a set of rules are the appearance of:

- conflicting rules,
- redundant rules,
- subsumed rules,
- unnecessary conditions in the body of a rule,
- circular chains of rules.

The issue of *rule validation* or *rule verification* is not elaborated in this thesis. Several books and special issues of journals on expert systems and knowledge based systems are dedicated to the validation and verification of knowledge based systems and expert systems (see [AL91, Gup91, Cul89, Fox92, pla93, prs93]). Other surveys and papers dedicated to the detection of inconsistencies and redundancies in rules (see [Cra87, GBP+92, LMP89, OO93, SC87, WS93]). Several tools that monitor the rule integrity are available, such as ONCOCIN (see [SSS88]), CHECK (see [NPLP87]), COVADIS (see [Rou88]), EVA (see [CCS90]), KB-REDUCER2 (see [Gin88]), the system proposed by Meseguer (see [Mes90]), KET (see [ET88]). There are several implementations of rule integrity checking systems available, although they are often not known as such, because they are often integrated in the database management system.

Here, we are especially interested in the field of data integrity in deductive database systems, although the results of this thesis are in fact applicable to any field concerning rules, such as expert systems, knowledge based systems or rule based systems. However, there may be a difference in the way these rules are handled, namely *data driven* or *goal driven*, but the analysis of inconsistencies and redundancies can in most cases be done in a common way.

This chapter introduces a method for checking the integrity constraints that should be used in a special module of the database management system, which we call $\mathcal{FICCS}$. $\mathcal{FICCS}$ is an acronym for $\mathcal{F}$act Integrity Constraint Checking $\mathcal{S}$ystem and is solely responsible for the manipulation and monitoring of constraints. The proposed method is based on the generation of meta-rules derived from the rules, inconsistency indicators and the general form of updates without accessing the fact base. These meta-rules are used for triggering some part of the integrity check depending on the specific updates in the transaction. These meta-rules are called inconsistency rules. In this chapter, we only show how $\mathcal{FICCS}$ should interact with a deductive database management system and elaborate the method based on inconsistency rules, which is the basis of $\mathcal{FICCS}$.

4.1 $\mathcal{FICCS}$; Deductive Database Systems and Integrity Constraint Checking

In this section the role that $\mathcal{FICCS}$ plays in a deductive database management system is sketched. $\mathcal{FICCS}$ is responsible for the optimized representation of the consistency check, but it is not responsible for the optimized evaluation of the consistency check. This representation of a consistency check contains a query, which is first optimized before it is evaluated by the query evaluator. The goal of the query evaluator is to handle queries as efficient as possible by using query optimization techniques mentioned in 1.3.2.2.

In $\mathcal{FICCS}$ inconsistency checks are represented as a set of inconsistency rules. Note that the goal of $\mathcal{FICCS}$ is to formulate a query that represents only that part of the database, which has to be checked after a transaction, i. e. , to present an optimized check. Further, $\mathcal{FICCS}$ is responsible for the maintenance of the set of inconsistency rules. So, when rules and inconsistency indicators are inserted, deleted or changed $\mathcal{FICCS}$ must take the necessary steps in order to keep the inconsistency rules up to date. FIGURE 4.1 illustrates the role of $\mathcal{FICCS}$ in the whole process of integrity constraint checking in a deductive databases. In CHAPTER 6 a detailed architecture of $\mathcal{FICCS}$ is given. In the remainder of this chapter the construction and optimization of inconsistency rules is elaborated.

4.1.1 Other Integrity Constraint Checking Systems

Some other systems that allow an integrity checking or integrity preserving system are the PRISM (see [SK84]) and Exegesis System (see [Sma88]).

In PRISM rules cannot be specified; so, there is only a distinction between facts and constraints. PRISM is a more object-oriented approach, which allows the natural specification of relations between objects, such as specialization and generilization of objects. Constraints, which are directly related to the data model are called *inherent constraints* here. For instance, in the object model an object in a class and an object in a subclass of that class must have some properties in common. Such constraints are incorporated in the data model. Further, in PRISM one distinguishes

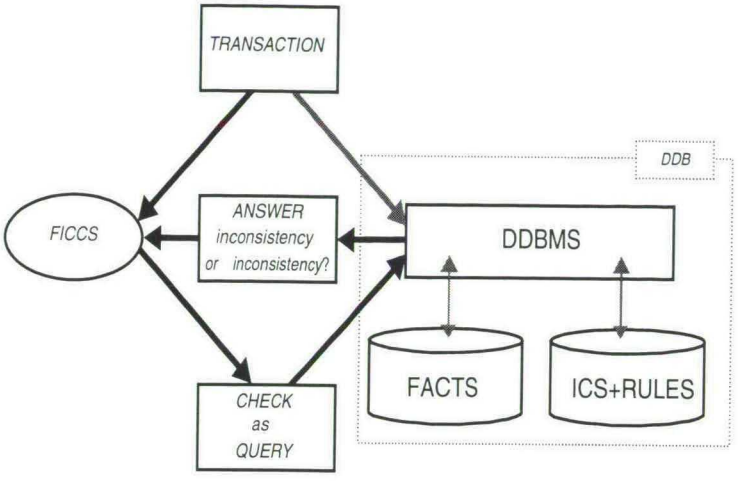


Figure 4.1: Architecture of a deductive database system with an integrity constraint checking component

explicit constraints and *implicit constraints*. Explicit constraints are constraints that are specified indepently from the data model, while implicit constraints are constraints that can be derived from the inherent constraints and the explicitly stated constraints. PRISM uses a declarative constraint language, in which the explicit constraints are represented. In the constraint specification one distinguishes a precondition, which determines at which circumstances this constraint has to be checked, an action, which must be executed when the precondition is fulfilled, and only if a postcondition, which comprises the actual constraint, is fulfilled. The action component of the constraint specification can be seen as the part that is responsible for the maintenance of the integrity of the fact base. Without the action component the constraint specifications can be seen as inconsistency rules in the relational case, where the head of an inconsistency rule corresponds to the precondition and the body of an inconsistency rule corresponds to the postcondition of the constraint specification in PRISM.

The Exegesis System, which is based on a deductive data model, checks the integrity of the database by the use of potential updates. In generating the potential updates it takes a top-down approach leaving out all subsumed potential updates generated in the process so far. Further, the Exegesis system allows a kind of default reasoning by the possibility of defining default rules. Such default rules are incorporated in the integrity checking process in Exegesis. Besides updates in relations, updating the rule base or constraint base, while keeping the fact base consistent, is also possible. However, the consistency of the rule base resp. constraint base itself is not checked. In Exegesis the evaluation of integrity constraints in the transaction has a higher priority

than the evaluation of constraints already present in the constraint base. Integrity constraints are represented as denials which can directly be processed by a resolution based query evaluator.

4.2 *FICCS*; Using Inconsistency Rules for Monitoring Consistency

The main feature of the proposed method based on inconsistency rules is that the consistency check itself is completely goal driven. The knowledge how an arbitrary update may influence inconsistency indicators is represented by so called *inconsistency rules*. These rules are meta-rules that are asserted to the deductive database. By the application of these rules, from any update the relevant instantiated inconsistency indicators, that have to be evaluated in the deductive database, are found in just one step. Therefore, it does not have the disadvantage of generating induced updates or potential updates that are not relevant to any inconsistency indicator. Hence, redundancy of the first type does not appear in the method based on inconsistency rules.

In 4.2.1 the advantages of the method based on inconsistency rules are shown, especially when it is compared to the methods based on induced and potential updates. It is shown that this improvement is fundamental.

4.2.1 Integrity Constraint Checking based on Inconsistency Rules

The new method is based on a new concept called *inconsistency rules*. These rules are constructed by using the rules and the inconsistency indicators in the deductive database. In the method based on inconsistency rules three phases are distinguished:

(i) *the generation phase*, in which the inconsistency rules from the rules and the inconsistency indicators in the database are derived,

(ii) *the compilation phase*, in which the inconsistency rules are optimized and redundancies are minimized, whereafter these rules are added to the database,

(iii) *the application phase*, in which only those inconsistency rules that are relevant to the update are triggered.

The inconsistency rules are derived from *inconsistency trees*, which in turn are derived from *potential update AND/OR trees*. For the moment, the more complex databases, in which negative database literals appear, are avoided. We elaborate the concept of potential update AND/OR trees, inconsistency trees and inconsistency rules by leaving out negative database literals in rules or inconsistency indicators first. Note that a deletion in such a database is not relevant to any inconsistency indicator. So, for the time being deletions will not influence the integrity of the database. In 5.1.2.1 negative database literals are introduced again.

4.2.1.1 Potential Update AND/OR Trees without Negation

Each potential update AND/OR tree is derived from some database literal relevant to an inconsistency indicator. So, from literals expressing some computation or comparison without accessing the database no potential update AND/OR trees are derived. In the next definition database literals are supposed to be positive until further notice.

DEFINITION 4.1 Let L be a database literal that appears in an inconsistency indicator. Literal L is the root of a *potential update AND/OR tree*, say T_L. L is called the *root literal* of T_L. We start with L as the first constructed node. Let $\mathcal{N}$ be a constructed node, then the following construction rules are applicable:

(i) If $\mathcal{N}$ is unifiable with the head of any rule, then the construction of T_L is a top-down construction which proceeds as follows:
Let $R : H \longleftarrow B_1 \wedge \cdots \wedge B_m$ be a rule, where H is a positive literal which is unifiable with $\mathcal{N}$ and where $B_1, \ldots, B_m$ are literals. Let σ be the most general unifier of $\mathcal{N}$ and H; then for each j the literal $B_j\sigma$ is an AND-node with respect to rule R of $\mathcal{N}$ only if it is not redundant. If the literal is redundant it is not part of the potential update AND/OR tree again. If more than one rule is applicable than for each rule there is an OR-branch for the literal $\mathcal{N}$, where each OR-branch ends in a group of related AND-nodes corresponding to the body of the applied rule.
A literal in the construction process is redundant if

- ➢ it is syntactically the same as some other node in the constructed potential update AND/OR tree so far, or
- ➢ it is syntactically the same as some other node in the constructed potential update AND/OR tree so far, except that both nodes only differ with respect to some variables that do not occur in the root literal.

(ii) If $\mathcal{N}$ is not unifiable with the head of any rule, then $\mathcal{N}$ does not have any child node, i. e., the construction process stops.

REMARK For each child node the construction rules are applied until none of these rules can be applied anymore. It may be obvious that in case of nonrecursive databases this construction process stops because rules can be applied only once and there are finitely many rules containing finitely many body literals. In the case of recursive databases the construction process also stops, as is pointed out in 5.3.1.

Note that for each database literal that is relevant to some inconsistency indicator a potential update AND/OR tree is created. Note that although from literals appearing in inconsistency indicators that correspond to evaluable predicates no potential update AND/OR trees are constructed. These literals still may be present as leafs in those trees.

EXAMPLE 4.1 Consider the database with the rules, facts and inconsistency indicator of EXAMPLE 2.4. Now, $parent(X, Y)$, which appears in the inconsistency indicator, is the root literal of the potential update AND/OR tree $T_{parent(X,Y)}$. There exist two OR-branches of $parent(X, Y)$,

i. e., a branch which ends in AND-node *mother*(X, Y) and a branch which ends in AND-node *father*(X, Y). Now, by applying the *mother*-rule to *mother*(X, Y) two related AND-nodes corresponding to the literals in the body of the *mother*-rule are derived, i. e., *husband*(Z, X) and *father*(Z, Y). An arc between the branches to *husband*(Z, X) and *father*(Z, Y) expresses the fact that they are related AND-nodes (see FIGURE 4.2 for the complete potential update AND/OR tree for *parent*(X, Y)).

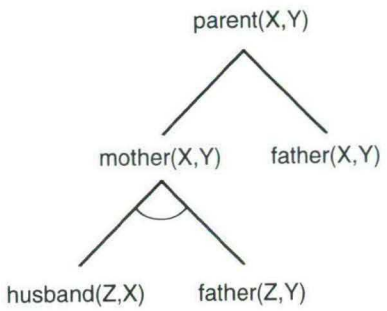


Figure 4.2: Potential update AND/OR tree for EXAMPLE 4.1

DEFINITION 4.2 Let II be an inconsistency indicator. A node in a potential update AND/OR tree of II is called an *updatable node* if it corresponds to an updatable relation.

Remember that we assumed that a node is updatable if and only if it corresponds to a base relation. Relations that are not updatable are not responsible for any change of the consistency of the database. Any success after the evaluation of an inconsistency indicator can only be caused via an updatable node. Therefore only the leaf nodes corresponding to base relations in the potential update AND/OR tree correspond to updatable nodes and are relevant for deriving the relevant instances of the inconsistency indicators, e. g., the leaf node *husband*(Z, X) in FIGURE 4.2. So, all nodes "between" the root and those leaf nodes are not updatable. FIGURE 4.3 shows how in our example the concepts of potential update AND/OR tree and inconsistency indicator interact. An update may instantiate a leaf node of some potential update AND/OR tree. By instantiating the leaf node of a potential update AND/OR tree the root literal of this potential update AND/OR tree is instantiated too. In FIGURE 4.3 the substitution $\{X/2, Z/1\}$ will instantiate *parent*(X, Y) in II_1. The instantiated root literal of this potential update AND/OR tree instantiates the inconsistency indicator. The instantiated inconsistency indicator is equal to the potential instance of the inconsistency indicator with respect to the instantiated root literal. So, the instantiated root literal is a potential update with respect to the update in the leaf node. This is the reason for calling these AND/OR trees *potential update* AND/OR trees.

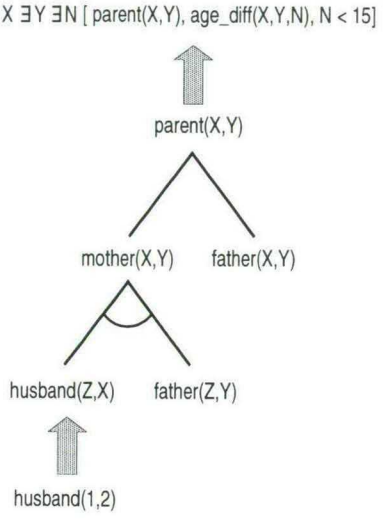


Figure 4.3: Updatable nodes in the potential update AND/OR tree for *parent* leading to II_1

4.2.1.2 Inconsistency Trees

Instantiation of inconsistency indicators by the updates in the transaction can be done in just one step. To express this one-step instantiation we construct *inconsistency trees*. They are defined using the definition of potential update AND/OR trees.

DEFINITION 4.3 Let II be an inconsistency indicator. An *inconsistency tree* (also called a *one-level inconsistency tree*, see [Sel95]) $\mathcal{T}_{II}$ is constructed as follows. The root of an inconsistency tree $\mathcal{T}_{II}$ is II. $\mathcal{N}$ is a child node of the root (i. e. , II) of $\mathcal{T}_{II}$ if it is an updatable node of a potential update AND/OR tree, T_L, for some literal L in II. From $\mathcal{N}$ no other nodes are derived.

Here, it is prohibited for a derived relation to be updatable. However, in the above definition only a strict distinction between updatable relations and relations which are not updatable is made. So, this definition does not exclude updates in views explicitly.

REMARK An instantiation of a node implied by some update now leads directly to a potential instance of the inconsistency indicator.

Although by DEFINITION 4.3 updating derived relations is not explicitly forbidden, here, the one-level inconsistency trees are constructed for base relations only, because we assumed that updatable nodes only contain base relations. So, by assumption, only the leaf nodes of potential update AND/OR trees correspond to base relations, and only these leafs are needed to build the inconsistency tree. The leaf nodes of an inconsistency tree are leaf nodes of a potential update AND/OR

tree and we therefore call all its leaf nodes *updatable nodes of the inconsistency tree*. The inconsistency tree is an ordinary tree, i. e. , there is no distinction between AND and OR nodes. Note that from each potential update AND/OR tree at least one inconsistency tree can be derived because, by definition, the root literal of a potential update AND/OR tree, say L, is a literal relevant to an inconsistency indicator. Because L may be relevant to other inconsistency indicators, several inconsistency trees may be derived from one and the same potential update AND/OR tree with L as root literal.

EXAMPLE 4.2 Consider EXAMPLE 4.1 with potential update AND/OR tree $T_{parent(X,Y)}$. From inconsistency indicator $II_1 : \exists X \exists Y \exists N[parent(X, Y), age\_diff(X, Y, N), N < 15]$ an inconsistency tree is derived. Because this inconsistency indicator contains two literals, i. e. , $parent(X, Y)$ and $age\_diff(X, Y, N)$, the inconsistency tree is built from the potential update AND/OR trees $T_{parent(X,Y)}$ and $T_{age\_diff(X,Y,N)}$. Each updatable node in the potential update AND/OR tree $T_{parent(X,Y)}$, i. e. , each node corresponding to the base relations *father* or *husband*, is a node in the inconsistency tree $\mathcal{T}_{II_1}$. Suppose we have the following rule which defines the *age_diff* relation

$$age\_diff(X, Y, N) \longleftarrow age(X, N1), age(Y, N2), N = N1 - N2,$$

then the corresponding complete inconsistency tree for II_1 can be found in FIGURE 4.4.

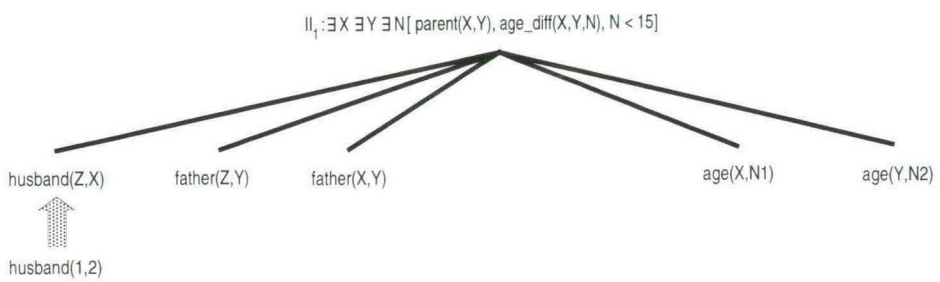


Figure 4.4: Inconsistency tree for II_1 in EXAMPLE 4.2

Note that *age* is considered here as a base relation. In real database applications the relation *age* as well as the relations *father* and *husband* may be derived from a database table as the next example illustrates. When this is the case, they are in fact derived relations and not updatable.

EXAMPLE 4.3 Suppose the relations *age*, *father* and *husband* could be derived from a `citizen` table of a register office database as follows:

R_1: $husband(X, Y) \longleftarrow$ `citizen`$(X, \_, Y, \_, \_, \_, \_, male)$

R_2: $father(X, Y) \longleftarrow$ `citizen`$(X, \_, \_, Y, \_, \_, \_, male)$, `citizen`$(Y, \_, \_, \_, \_, \_, \_, \_)$

R_3: $age(X, N) \longleftarrow$ `citizen`$(X, \_, \_, \_, \_, D, \_, \_)$, $date\_to\_age(D, N)$

where `citizen`(_, _, _, _, _, _, _, _) corresponds to a table of which the first argument is the citizen's identifier, the third argument is the identifier of the one he or she is married with, the fourth argument is an identifier of a child of that citizen, the sixth argument is the citizen's date of birth and the last argument is for the citizen's sex. All other arguments are not relevant and therefore are not specified here. Note that *date_to_age* is an evaluable predicate and therefore not updatable. So, the leaf nodes of the inconsistency tree would only contain the base relation `citizen`.

However, the relations *husband*, *father* and *age*, are considered as base relations in most cases, unless stated otherwise.

4.2.1.3 Inconsistency Rules

From the inconsistency trees inconsistency rules are constructed. Such rules are meta expressions which comprise the check that has to be performed for all possible kinds of updates to updatable relations. For, updates of relations for which no inconsistency rules exist no check is required.

The next definition shows how the inconsistency rules are derived from inconsistency trees.

DEFINITION 4.4 Let II be an inconsistency indicator and let A be a leaf node of the inconsistency tree $\mathcal{T}_{II}$. Then $Q_A[\mathit{inconsistent}(A) \Leftarrow II']$ is called an *inconsistency rule*, where II' is the existential quantified II in which all existential quantifiers that appears in A are removed and Q_A is the existential quantification of each variable in A and each free variable in II'.

EXAMPLE 4.4 Let II be $\exists X \exists Y[a(X, Y), b(Y)]$ and let A be equal to $c(Z, X)$. Then the inconsistency rule is defined as:

$$\exists X \exists Z[\mathit{inconsistent}(c(Z, X)) \Leftarrow \exists Y[a(X, Y), b(Y)]]$$

Note that A is an updatable node and therefore by assumption that A is a base literal.

From this point we denote the inconsistency indicators and inconsistency rules without existential quantifiers. In that case, the inconsistency rule derived from an inconsistency indicator II without quantification and a leaf node A is equal to $\mathit{inconsistent}(A) \Leftarrow II$. Hence, in our example the inconsistency rule is equal to $\mathit{inconsistent}(c(Z, X)) \Leftarrow a(X, Y), b(Y)$. Note that there cannot be any misunderstanding about the scope of the quantifiers, because in this thesis all formulas are rectified and all inconsistency indicators are closed. For each inconsistency indicator all derivable inconsistency rules are asserted to the database. These are meta-rules expressing the goals, which represent the checks, that have to be evaluated after a certain update of the database. Note that the evaluation, in the case without negation, must take place in the updated database.

DEFINITION 4.5 Let T be a transaction. Let $IR : \mathit{inconsistent}(A) \Leftarrow II$ be an inconsistency rule. If an update U in T that is unifiable with A, then we say that U influences IR and that IR is applicable to U. We say that the application of IR to U is true (resp. false) if $II\sigma$ is true (resp. false) in the updated database, where σ is the most general unifier of A and U.

It is sufficient to evaluate only those instances of inconsistency indicators which are derived from the update and the relevant leaf nodes of inconsistency trees. If such an instance is true, then the update is rejected. If not, the updated database is consistent. This is exactly what happens when inconsistency rules are applied that are influenced by the transaction.

PROPOSITION 9 *Let D be a consistent database and T a transaction. Then* D_T *is* consistent *iff each application of an inconsistency rule to each update in the transaction is false in* D_T.

Proof:
From PROPOSITION 8 one can see that the truth values of potential instances of inconsistency indicators in D_T are relevant for concluding that D_T is consistent. Therefore with PROPOSITION 8 in order to prove this proposition it is sufficient to prove that the applications of inconsistency rules lead to precisely those potential instances of inconsistency indicators, which were needed to check the consistency of D_T.

Suppose there exists an inconsistency rule $inconsistent(A) \Leftarrow II$ that is applicable to an update U in T. This implies that A is an updatable node of an inconsistency tree $\mathcal{T}_{II}$ and A is unifiable with U. Let σ be a most general unifier of A and U. Then $A\sigma$ leads to a potential update $L\sigma$, where L is the root literal of the potential update AND/OR tree that is relevant to II and A is a leaf node of that tree. The simplified instance of II with respect to $L\sigma$, i.e., $II\sigma$, is a *potential instance* of II. Conversely, from a potential instance of an II an application of an inconsistency rule is found. This completes the proof of this proposition. ❑

4.2.1.4 Redundancy in the Method based on Inconsistency Rules

In CHAPTER 3 we showed that the existing methods all have to deal with redundancies of one or more types. This is also true for the method based on inconsistency rules although it is already in most cases an improvement of existing methods based on induced updates and potential updates (see [Sel95]). This section presented a method based on inconsistency rules that does not contain the redundancy of the first type, like the method of potential updates, which is already an improvement compared to the method based on induced updates (see the remark in 3.2.2). However, similar to the method based on potential updates, the method based on inconsistency rules presented so far still contains redundancy, namely redundancy of the second and third kind. Redundancy of the second type is not present in the method based on induced updates (see the remark in 3.3).

In the next section, the method based on inconsistency rules is adjusted in order to combine the advantages of both the method based on potential updates and the method based on induced updates, hence avoiding the redundancy of the second kind as well. It even goes beyond the advantages of both methods, because redundancy of the third type is avoided too. In the next section this revised method based on inconsistency rules will be elaborated.

4.2.2 Integrity Constraint Checking based on Revised Inconsistency Rules

Inconsistency indicators played a crucial role in the method based on inconsistency rules presented above, because the body of an inconsistency rule corresponds directly to an inconsistency indicator. In order to avoid redundancy of the second and third type, inconsistency indicators have to be changed into revised inconsistency rules, which describe more accurately which part of the database is affected by an update.

4.2.2.1 Advantages of Revised Inconsistency Rules

In order to illustrate the advantages of revised inconsistency rules informally, consider EXAMPLE 2.4. Suppose the update to this database is *husband*$(1, 2)$ and let

$$parent(X, Y), age\_diff(X, Y, N), N < 15$$

be an inconsistency indicator. Following the construction of the inconsistency rules in the previous section the derived inconsistency rule with respect to the relation *husband* has the following form:

$$inconsistent(husband(Z, X)) \Leftarrow parent(X, Y), age\_diff(X, Y, N), N < 15.$$

This means that whenever the inconsistency rule is applied to the update *husband*$(1, 2)$, the evaluation in the updated database of the instantiated inconsistency indicator

$$parent(2, Y), age\_diff(2, Y, N), N < 15$$

is necessary. In fact, what we really want to know is if there exists a *parent* in the new database state, which was not present in the previous database state, for which the age difference to the parent's children is less than 15. Note that the update *husband*$(1, 2)$ only changes the database through the second *parent*-rule. In other words, only new mothers can contribute to the change of the *parent*-relation. But when evaluating the instantiated indicator, the subgoal *parent*$(2, Y)$ will try to find all parents of this form; so, the *mother*- as well as the *father*-part of the *parent*-rule is searched. But it is known from the update that the *father*-relation has not changed. The idea is to incorporate this knowledge into the inconsistency rule. In order to do so, the relation *parent* is unfolded until the update of concern is met. The literal *parent*(X, Y) in the inconsistency rule is replaced by an expression which gives a precise description of the change in *parent*. In general, if *husband*(Z, X) is an update for some binding of Z and X, the *mother*-rule states that *mother*(X, Y) is a new instance if there exist fathers of the format *father*(Z, Y) in the database. So, instances of *father*(Z, Y) will give new instances of *mother*(X, Y) and consequently new instances of *parent*(X, Y). So, only an instance of *father*(Z, Y) determines a new instance of *parent*. Therefore, in our example in the inconsistency rule with respect to *husband*, *parent*(X, Y) can be replaced by *father*(Z, Y). The revised inconsistency rule is:

$$inconsistent(husband(Z, X)) \Leftarrow father(Z, Y), age\_diff(X, Y, N), N < 15.$$

These revised inconsistency rules can be derived easily from the revised inconsistency trees by taking each leaf node of the revised inconsistency tree as the argument in the head of the revised

inconsistency rule and the root of the revised inconsistency tree as the body of the revised inconsistency rule. Before giving the formal definition of revised inconsistency tree and revised inconsistency rule a concept called update expression is introduced, which is used in those formal definitions.

4.2.2.2 Update Expressions

In the previous definition of inconsistency tree, the root of such a tree is an inconsistency indicator. For each inconsistency indicator exactly one tree exists. Now, for each updatable node in the previously defined inconsistency tree, a separate revised inconsistency tree is constructed by using once again the potential update AND/OR tree. Now, the root of a revised inconsistency tree corresponds to an optimized inconsistency indicator. It will check only that part of the database that is influenced by the updatable node. To clarify this situation the following definitions and examples are helpful.

DEFINITION 4.6 Let C and D be literals appearing in some potential update AND/OR tree T_L, in which C is ancestor of D. A conjunction of literals is collected from T_L starting with D as the current node and the empty conjunction. The collecting process proceeds as follows.

- ➢ Let D' be the parent node of the current node, then collect all child AND-nodes related to the current node, excluding the current node, and add these nodes to the current literal set, resulting in the literal set $S_{D'}$.

Proceed this process with D' as the current node and $S_{D'}$ as the current literal set.

Continue this algorithm until C is reached. By Δ_D^C we denote the conjunction of all collected AND-nodes in S_C in order of derivation, or $true$ if $S_C = \emptyset$.

DEFINITION 4.7 Let II be an inconsistency indicator, let L be a literal in II and let $\mathcal{N}$ be an updatable node from the potential update AND/OR tree T_L. Then we call $\Delta_{\mathcal{N}}^L$ the *update expression* of L by $\mathcal{N}$.

EXAMPLE 4.5 Consider EXAMPLE 4.2. The update expressions with respect to $parent(X, Y)$ are

$$\Delta^{parent(X,Y)}_{husband(Z,X)} = father(Z, Y)$$
$$\Delta^{parent(X,Y)}_{father(Z,Y)} = husband(Z, X)$$
$$\Delta^{parent(X,Y)}_{father(X,Y)} = true$$

for which $II_{parent(X,Y)}$ is equal to $age\_diff(X, Y, N)$, $N < 15$. The update expressions with respect to $age\_diff(X, Y, N)$ are

$$\Delta^{age\_diff(X,Y,N)}_{age(X,N1)} = age(Y, N2), N = N1 - N2$$
$$\Delta^{age\_diff(X,Y,N)}_{age(Y,N2)} = age(X, N1), N = N1 - N2$$

for which $II_{age\_diff(X,Y,N)}$ is equal to $parent(X, Y)$, $N < 15$.

REMARK $\Delta^L_{\mathcal{N}}$, which replaces L in II and depends on a particular updatable node $\mathcal{N}$, expresses the computation that has to be performed in order to derive the induced updates with respect to the relation corresponding to L when the relation corresponding to $\mathcal{N}$ is updated. For each of these induced updates the remainder of the II is (partially) instantiated and evaluated. So, the update expression determines the change of L after an update in $\mathcal{N}$.

4.2.2.3 Revised Inconsistency Rules

In this section, update expressions are used for defining Revised Inconsistency Rules. In fact, update expressions are used to alter the inconsistency indicators to get a precise description of what has to be checked after a certain update.

DEFINITION 4.8 Let II be an inconsistency indicator, let L be a literal in II and let $\mathcal{N}$ be an updatable node from the potential update AND/OR tree T_L. The expression that is derived from II by replacing L by $\Delta^L_{\mathcal{N}}$ is called a *revised inconsistency indicator* with respect to L and $\mathcal{N}$. This revised inconsistency indicator is denoted by $II(L, \mathcal{N})$.

DEFINITION 4.9 Let II be an inconsistency indicator, let L be a literal in II. We call the expression that is derived from II by leaving out L from the conjunction, which is denoted by II_L, the *remainder* of II with respect to L.

REMARK The revised inconsistency indicator in DEFINITION 4.8 is expressed by:

$$II(L, \mathcal{N}) = \Delta^L_{\mathcal{N}}, II_L$$

EXAMPLE 4.6 Consider the situation in EXAMPLE 4.2. The revised inconsistency indicator with respect to *parent*(X, Y) and the node *husband*(Z, X) is derived from the inconsistency indicator by replacing *parent*(X, Y) by the only collected node *father*(Z, Y). Note that when in the collection process *mother*(X, Y) is the current node, the next node is *parent*(X, Y), but there are no AND-nodes collected.

This example comprises exactly the description of the inconsistency check we want to perform, as we noted in the informal introduction at the beginning of this section.

DEFINITION 4.10 Let II be an inconsistency indicator and $\mathcal{T}_{II}$ the corresponding inconsistency tree. For each leaf node $\mathcal{N}$ in $\mathcal{T}_{II}$, which is by definition an updatable node of a potential update AND/OR tree, T_L, for some literal L in II, a separate *revised inconsistency tree* $\mathcal{T}_{II(L,\mathcal{N})}$ is defined. $\mathcal{T}_{II(L,\mathcal{N})}$ has as root the *revised inconsistency indicator* with respect to L and $\mathcal{N}$ and $\mathcal{N}$ as (the only) child node.

EXAMPLE 4.7 Consider the update expressions derived in EXAMPLE 4.5. From these update expressions and the proper remainders of II the revised inconsistency indicators are derived as in the remark above. As a result five revised inconsistency indicators are derived, which are called II_{1a}, II_{1b}, II_{1c}, II_{1d} and II_{1e} respectively. Hence, five revised inconsistency trees namely $\mathcal{T}_{II_{1a}}$, $\mathcal{T}_{II_{1b}}$, $\mathcal{T}_{II_{1c}}$, $\mathcal{T}_{II_{1d}}$ and $\mathcal{T}_{II_{1e}}$ are derived. These revised inconsistency trees with respect to II_1 of

EXAMPLE 4.2 are presented in FIGURE 4.5. Note that $N = N1 - N2, N < 15$ is abbreviated to $N1 - N2 < 15$.

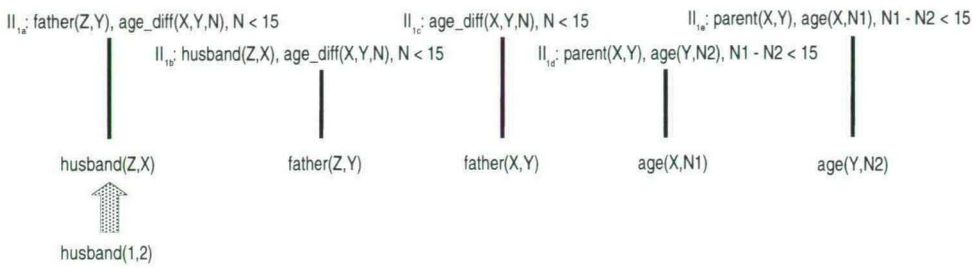


Figure 4.5: Revised inconsistency trees for EXAMPLE 4.7

Now, revised inconsistency rules are derived from revised inconsistency trees in a trivial manner.

DEFINITION 4.11 Let $\mathcal{T}_{II(L,C)}$ be a revised inconsistency tree, where C is its leaf node. Then $inconsistent(C) \Leftarrow II(L, C)$ is called a *revised inconsistency rule*.

EXAMPLE 4.8 The five revised inconsistency rules that are derived from EXAMPLE 4.7 are:

$$inconsistent(husband(Z, X)) \Leftarrow father(Z, Y), age\_diff(X, Y, N), N < 15$$

$$inconsistent(father(Z, Y)) \Leftarrow husband(Z, X), age\_diff(X, Y, N), N < 15$$

$$inconsistent(father(X, Y)) \Leftarrow age\_diff(X, Y, N), N < 15$$

$$inconsistent(age(X, N1)) \Leftarrow parent(X, Y), age(Y, N2), N1 - N2 < 15$$

$$inconsistent(age(Y, N2)) \Leftarrow parent(X, Y), age(X, N1), N1 - N2 < 15$$

Note that the definition of Δ guarantees the following property:

$$\Delta_D^B = \Delta_C^B, \Delta_D^C$$

for literals B, C and D in some potential update AND/OR tree, where each of the Δ's are defined and ',' represent the operation of conjunction between the two operands. This property shows that an adjustment of the database schema will lead to a natural adjustment of Δ. For instance, the adjustment of the definition of a predicate which appears in C does not imply a complete re-computation of Δ_D^B, but is constrained to Δ_D^C.

EXAMPLE 4.9 Consider the database as described by EXAMPLE 4.3 in which the relations *age*, *father* and *husband* are defined in terms of a `citizen` table. The following properties hold:

$$\Delta^{husband(Z,X)}_{\texttt{citizen}(Z,\_,X,\_,\_,\_,\_,male)} = true$$

$$\Delta^{father(X,Y)}_{\texttt{citizen}(X,\_,\_,Y,\_,\_,\_,male)} = \texttt{citizen}(Y,\_,\_,\_,\_,\_,\_)$$

$$\Delta^{father(X,Y)}_{\texttt{citizen}(Y,\_,\_,\_,\_,\_,\_,\_)} = \texttt{citizen}(X,\_,\_,Y,\_,\_,\_,male)$$

$$\Delta^{age(X,N)}_{\texttt{citizen}(X,\_,\_,\_,\_,D,\_,\_)} = date\_to\_age(D,N)$$

Together with the results of EXAMPLE 4.7 the update expressions can be derived by composing related Δ's, which results in the following revised inconsistency rules:

$$inconsistent(\texttt{citizen}(Z,\_,X,\_,\_,\_,\_,male)) \Leftarrow$$
$$father(Z,Y), age\_diff(X,Y,N), N < 15$$

$$inconsistent(\texttt{citizen}(Z,\_,\_,Y,\_,\_,\_,male)) \Leftarrow$$
$$\texttt{citizen}(Y,\_,\_,\_,\_,\_,\_,\_), husband(Z,X), age\_diff(X,Y,N), N < 15$$

$$inconsistent(\texttt{citizen}(Y,\_,\_,\_,\_,\_,\_,\_)) \Leftarrow$$
$$\texttt{citizen}(X,\_,\_,Y,\_,\_,\_,male), age\_diff(X,Y,N), N < 15$$

$$inconsistent(\texttt{citizen}(Z,\_,\_,Y,\_,\_,\_,male)) \Leftarrow$$
$$\texttt{citizen}(Y,\_,\_,\_,\_,\_,\_,\_), husband(Z,X), age\_diff(X,Y,N), N < 15$$

$$inconsistent(\texttt{citizen}(Y,\_,\_,\_,\_,\_,\_,\_)) \Leftarrow$$
$$\texttt{citizen}(X,\_,\_,Y,\_,\_,\_,male), age\_diff(X,Y,N), N < 15$$

$$inconsistent(\texttt{citizen}(X,\_,\_,\_,\_,D,\_,\_)) \Leftarrow$$
$$date\_to\_age(D,N1), parent(X,Y), age(Y,N2), N1 - N2 < 15$$

$$inconsistent(\texttt{citizen}(Y,\_,\_,\_,\_,D,\_,\_)) \Leftarrow$$
$$date\_to\_age(D,N2), parent(X,Y), age(Y,N1), N1 - N2 < 15$$

Note in this example the use of the variables in the inconsistency rules. Some of the variables are kept unspecified, while others are named. Now, several definitions are stated in order to distinguish the role of these variable.

DEFINITION 4.12 Let IR be an inconsistency rule. Let X be a distinguished variable of IR. X is called an *update variable* if it appears also in the body of the rule.

DEFINITION 4.13 Let IR be an inconsistency rule. Let Y be a nondistinguished variable of IR. Y is called a *connection variable* if it appears in at least two literals of the body in the rule.

DEFINITION 4.14 Let IR be an inconsistency rule. A variable Z is called *irrelevant* if it is neither an update variable nor a connection variable of IR.

EXAMPLE 4.10 Consider the revised inconsistency rule

$$inconsistent(\texttt{citizen}(Z,\_,X,\_,\_,\_,\_,male)) \Leftarrow$$
$$father(Z,Y), age\_diff(X,Y,N), N < 15.$$

Variables X and Z are update variables, Y and N are connection variables, while all irrelevant variables remain unspecified.

Note that the update variables determine the evaluation order of the body literals, that the connection variables are instantiated by instantiation of update variables and that irrelevant variables do not play a relevant role in the consistency check. We suppose that the body literals in revised inconsistency rules are delivered in the optimal evaluation order. More on the influence of the evaluation order to the overall query can be found in [Ull88].

In CHAPTER 3 we saw that all existing methods have to deal with redundancies of one or more types. This is also true for the unrevised method based on inconsistency rules although it is already in most cases an improvement of the existing methods based on induced updates and potential updates (see [Sel94, Sel95]). This section presented a revised method based on inconsistency rules that

- does not contain the redundancy of the first type, because in methods based on inconsistency rules no inconsistency rule exists for updates not relevant to any inconsistency indicator. So, such updates will not lead to any action in our method as we might have expected.
- reduces the redundancy of the third type, because only branches with changes are evaluated.
- can minimize the redundancy of the second type by an optimal order of the subgoals in the body of the inconsistency rules.

The optimal order of the subgoals can be determined by an query optimizer, but *FICCS* can deliver also revised inconsistency rules in which an optimal order of the subgoals is determined. Besides the nice properties of the presented method concerning redundancies, the method based on inconsistency rules can be extended in several ways. For instance, we could introduce negation or recursion into our language, or we could allow updates of rules and/or constraints or even replacements in our transactions. These extensions are elaborated in the next chapter.

References

[AL91] M. AYEL AND J. P. LAURENT, editors. *Validation, Verification and Test of Knowledge-Based Systems*. John Wiley & Sons Ltd., Chichester, England, 1991.

[CCS90] C. CHANG, J. COMBS AND R. STACHOWITZ. A Report on the Expert Systems Validation Associate(EVA). Technical report, Lockheed Missiles and Space Company, 1990.

[Cra87] B. J. CRAGUN. A Decision-Table-Based Processor for Checking Completeness and Consistency in Rule-Based Expert Systems. *International Journal of Man-Machine Studies*, 26:633–648, 1987.

[Cul89] C. CULBERT, editor. *Expert Systems with Applications. Special issue: Verification and Validation of Knowledge-Based Systems*, volume 1. 1989.

[ET88] L. ESFAHANI AND F. TESKEY. A Self Modifying Rule-Eliciter. In *Proceedings of the European Knowledge Acquisition*, pages 16.1–16.16, 1988.

[Fox92] J. FOX, editor. *The Knowledge Engineering Review. Special issue: Knowledge Base Verification*, volume 7. 1992.

[GBP+92] P. GROGONO, A. BATAREKH, A. PREECE, R. SHINGHAL AND C. SUEN. Expert System Evaluation Techniques: A Selected Bibliography. *Expert Systems with Applications*, pages 227–239, 1992.

[Gin88] A. GINSBERG. Knowledge-Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency Redundancy. In *Proceedings of the Seventh National Conference on AI; AAAI'88*, pages 585–589, August 1988.

[Gup91] U. GUPTA, editor. *Validating and Verifying Knowledge-Based Systems*. IEEE Computer Society Press, Los Alamitos, California, 1991.

[LMP89] B. LOPEZ, P. MESEGUER AND E. PLAZA. Knowledge-Based Systems Validation: A State of the Art. *AI Communication*, 3(2):58–72, 1989.

[Mes90] P. MESEGUER. A New Method for Checking Rule Base for Inconsistency: A Petri Net Approach. Technical Report 90-6, GRIAL, Blanes, 1990.

[NPLP87] TIN A. NGUYEN, WALTON A. PERKINS, THOMAS J. LAFFEY AND DEANNE PECORA. Knowledge Base Verification. *AI Magazine*, 2(8):69–75, Summer 1987.

[OO93] R. M. O'KEEFE AND D. E. O'LEARY. Expert System Verification and Validation: a Survey and Tutorial. *Artificial Intelligence Review*, 7:3–42, 1993.

[pla93] Special Issue: Validation & Verification of Knowledge-Based Systems. 8(3), 1993.

[prs93] The International Journal of Expert Systems: Research and Applications. Special issue: Verification & Validation. 6(2-3), 1993.

[Rou88] M. C. ROUSSET. On the Consistency of Knowledge Bases: the Covadis System. In *Proceedings of the Eighth European Conference on Artificial Intelligence; ECAI'88*, pages 79–84, 1988.

[SC87] ROLF A. STACHOWITZ AND JACQUELINE B. COMBS. Validation of expert Systems. In *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, volume 1, pages 689–695, 1987.

[Sel94] RON R. SELJÉE. Integrity Constraint Checking for Updates in Deductive Databases; A Different Approach. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 52:23p, 1994.

[Sel95] RON R. SELJÉE. A New Method for Integrity Constraint Checking in Deductive Databases. *Data & Knowledge Engineering*, 15(1):63–102, March 1995.

[SK84] ALLAN SHEPHERD AND LARRY KERSCHBERG. PRISM: a Knowledge Based System for semantic Integrity Specification and Enforcement in Database Systems. In BEATRICE YORMARK, editor, *ACM SIGMOD RECORD*, volume 14-2, pages 307–315, Boston, MA, June 1984. ACM Press.

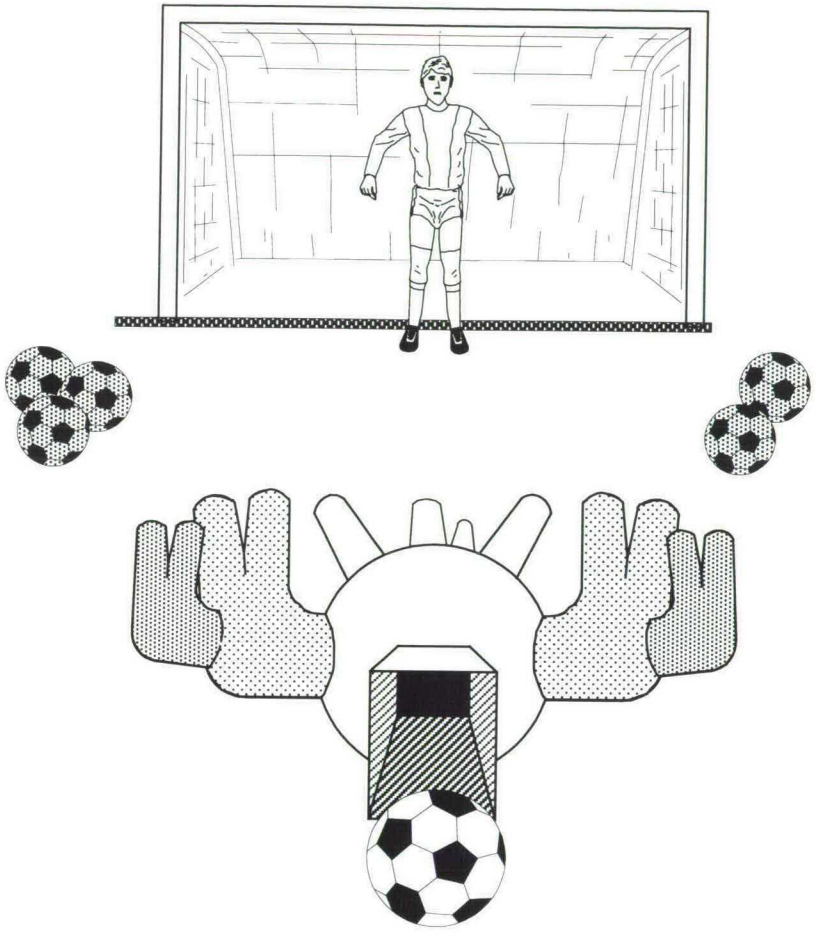
[Sma88] CAROL SMALL. The Implementation of the Exegesis System. *The Computer Journal*, 31(2):125–132, 1988.

[SSS88] M. SUWA, A. C. SCOTT AND E. H. SHORTLIFFE. An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System. *AI Magazine*, 3(4):16–21, Fall 1988.

[Ull88] JEFFREY D. ULLMAN. The Complexity of Ordering Subgoals. In *Proceedings of the Seventh ACM SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 74–81, March 1988.

[WS93] PING WU AND STANLEY Y. W. SU. Rule Validation Based on Logical Deduction. In BHARAT K. BHARGAVA, TIMOTHY W. FININ AND YELENA YESHA, editors, *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 164–173, Washington, DC, USA, November 1993. ACM Press.

". . . and it could cause an inconsistency like previous updates did . . ."



Chapter 5

Extensions of *FICCS*

One of the main advantages of integrity checking by using inconsistency rules is that this method can easily be extended in several directions. We can extend our data model, we can allow more general updates in our transactions like rule updates, constraint updates or replacements, we can add negation to our language or allow recursion in our rules and inconsistency indicators. These extensions can be derived by adjusting the inconsistency rules without changing other parts of the database. In this chapter, some of these extensions are elaborated and can be incorporated in the implementation directly, while others are just described and need some further research.

5.1 Extended Datamodel

In this thesis, the emphasis lies on integrity checking combined with deduction rules defined on top of a purely relational datamodel. However, the creation of revised inconsistency rules for deductive rules or even other kinds of rules, such as production rules, defined on top of other datamodels, such as the object-oriented datamodel seems possible. For a comparison of deductive and object-oriented database systems we refer to [Ull91].

5.1.1 Integrity Checking in Other Datamodels

In [Deß93] a survey is given of semantic integrity constraints in all kinds of systems, such as *active database systems*, *deductive database systems*, *object-oriented database systems*, etc.. It presents the interaction between integrity constraints and knowledge bases in a more architectural way. Although [Deß93] does not give a detailed description of the way the integrity has to be checked, it does give an outline of the considerations which are relevant for checking or enforcing integrity constraints in knowledge bases in general.

This thesis shows that integrity checking in deductive databases, when based on inconsistency rules, is a natural extension of the relational case. As a result, integrity checking in object-oriented systems and in extensions like deductive object-oriented systems is comparable to integrity checking in deductive databases. Inconsistency rules can easily be transformed into methods. The body of a revised inconsistency rule, i. e. , an evaluation of the related revised inconsistency indicator,

must be a method called *update* of an object, which is named after the predicate name in the *inconsistent* predicate in the head of the inconsistency rule. So, when that object is updated the method *update* is fired, in which case the proper revised inconsistency indicator must be evaluated.

From [Deß93], [JJ91], [JK90], [Jeu92] and [STW93] one can conclude that integrity constraints in knowledge bases, such as *active databases*, *deductive databases* or *object-oriented databases*, are handled similarly. Therefore, the application of revised inconsistency rules seems to be a powerful tool for handling integrity constraints in all kinds of knowledge bases.

5.1.2 Extended Language

In the previous part of this thesis the formulas in which we specify rules and constraints are universally quantified, function free and without negation. In this section, we will study the consequences for the method based on inconsistency rules when negation is introduced in our formulas. Further, we will allow a restricted form of existential quantifiers in constraints. Introducing functions can cause difficulties for which we will refer to the literature. Reasoning with equality, i. e. , stating that terms are the same, can be handled trivially when function symbols are forbidden in our language. When the deductive system is able to reason with equality, it can be incorporated in our language as well, because such an extension is due to the capabilities of the deductive database system and not a gain by the proposed method. Hence, the extension with equality is not elaborated here. For an elaborated research on equality one is invited to read [Kog95], in which it is shown that reasoning with equality between terms with function symbols is also possible.

5.1.2.1 Negation in the Method based on Inconsistency Rules

In this section, negation in database clauses and inconsistency indicators is permitted. However, because of the asymmetry in the handling of negation in the updated database, as noted in the remark after DEFINITION 2.9 and DEFINITION 2.17, this asymmetry influences the construction of (revised) inconsistency rules a great deal. First, the definition of a potential update AND/OR tree is adjusted before adjusting the definition of inconsistency rules.

DEFINITION 5.1 Let L be a database literal that appears in an inconsistency indicator. Literal L is the root of a *potential update AND/OR tree*, say T_L. L is called the *root literal* of T_L. We start with L as the first constructed node. Let $\mathcal{N}$ be a constructed node, then the following construction rules are applicable:

(i) If $\mathcal{N}$ is unifiable with the head of any rule or its negation, then the construction of T_L is a top-down construction which proceeds as follows.
Let $R : H \longleftarrow L_1 \wedge \cdots \wedge L_m$ be a rule, where H is an atom and where $L_1, \ldots, L_m$ are literals.

➢ If $\mathcal{N}$ is unifiable with H and σ is the most general unifier of $\mathcal{N}$ and H; then for each j the literal $L_j\sigma$ is an AND-node with respect to rule R of $\mathcal{N}$ unless it is redundant.

- If $\mathcal{N}$ is unifiable with $\neg H$ and σ is the most general unifier of $\mathcal{N}$ and $\neg H$; then for each j the literal $\neg L_j\sigma$ is an AND-node with respect to rule R of $\mathcal{N}$ unless it is redundant.

If the literal is redundant it is not asserted as a node to the potential update AND/OR tree again. If more than one rule is applicable than for each rule there is an OR-branch for the literal $\mathcal{N}$, where each OR-branch ends in a group of related AND-nodes corresponding to the body of the applied rule.
A literal in the construction process is redundant if

- it is syntactically the same as some other node in the constructed potential update AND/OR tree so far, or
- it is syntactically the same as some other node in the constructed potential update AND/OR tree so far, except that both nodes only differ with respect to some variables that do not occur in the root literal.

(ii) If $\mathcal{N}$ is not unifiable with the head of any rule or its negation, then $\mathcal{N}$ does not have any child node, i.e., the construction stops.

EXAMPLE 5.1 Let D be a database with the following rule base:

RULES

R_1: $a(X) \longleftarrow b(X, Y), \neg c(Y, Z)$

R_2: $a(X) \longleftarrow f(X, Y), g(Y, X)$

R_3: $c(Y, Z) \longleftarrow d(V, Y), e(Z), \neg b(V, Y)$

R_4: $c(Y, Z) \longleftarrow b(Y, Z)$

Then the potential update AND/OR tree with root $a(X)$ is the tree presented in FIGURE 5.1.

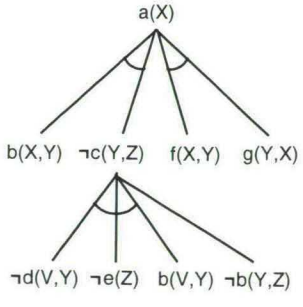


Figure 5.1: Potential update AND/OR tree with negative literals

The definition of Δ is adjusted in order to handle negation as well. There exists only a slight difference in the way the literals are collected in deriving Δ in case of negation; namely, the parent literal is replaced unless its parent node is negative, in which case the parent node is in some cases added to the current literal set.

DEFINITION 5.2 Let C and D be literals appearing in some potential update AND/OR tree T_L, in which C is an ancestor of D. A conjunction of literals is collected from T_L starting with D as the current node and the empty conjunction. The collecting process proceeds as follows.

➢ Let D' be the parent node of the current node,

(i) if D' is a positive literal, then collect all child AND-nodes related to the current node, excluding the current node. Add the collected nodes to the current literal set,

(ii) if D' is a negative literal, then negate all child AND-nodes related to the current node and collect them, excluding the current node, and mark them with an asterisk. Further, include D' only if D' has more than one OR-node or do not include D' otherwise. Add these nodes to the current literal set,

which results in the literal set $S_{D'}$ with respect to C and D.

Repeat these steps with D' as the current node and $S_{D'}$ as the current literal set until C is reached.

By Δ_D^C we denote the conjunction of all collected AND-nodes in S_C in order of derivation, where marked literals must be evaluated in the database state before the update of the updatable node takes place.

The adjusted definition for Δ and its consequences for update expressions and revised inconsistency indicators is illustrated by the following example.

EXAMPLE 5.2 Consider EXAMPLE 5.1 with the addition of the following inconsistency indicator:

$$a(X), \neg d(X, X)$$

The revised inconsistency indicator $II_1(a(X), \neg d(V, Y))$ with respect to $a(X)$ and node $\neg d(V, Y)$ is derived as follows. Start with the node $\neg d(V, Y)$ of the potential update AND/OR tree as presented in EXAMPLE 5.1 as current node and an empty literal set. After the first step, we have

$$S_{\neg c(Y,Z)} = \{e(Z)^*, \neg b(V, Y)^*, \neg c(Y, Z)\}$$

as the current literal set with respect to $a(X)$ and node $\neg d(V, Y)$. After this step, the current node is $\neg c(Y, Z)$, for which $a(X)$ is the parent node. The final literal set $S_{a(X)}$ with respect to $a(X)$ in II_1 and $\neg d(V, Y)$ is the set

$$\{b(X, Y), e(Z)^*, \neg b(V, Y)^*, \neg c(Y, Z)\}.$$

REMARK $S_{\neg c(Y,Z)}$ expresses the evaluation that has to be performed in order to derive induced deletions $c(Y, Z)$, in which case the marked literals have to be evaluated in the previous database state. Further, for each induced deletion its effectiveness is checked, in other words: "does $\neg c(Y, Z)$ hold for each substitution of Y and/or Z that is found in the updated database".

This is exactly according to the remarks on induced deletions and effective updates before DEFINITION 2.3 and after DEFINITION 2.9, respectively. In order to distinguish literals being evaluated in the current database state and literals being evaluated in the new database state two meta-predicates old_D and $new_{D,T}$ are defined, which express whether its argument holds in the current database D resp. the updated database D_T given transaction T. When the database D and the transaction T are clear from the context or unspecified, the predicates *old* and *new* are used without their subscripts. In both cases the argument is a literal or a conjunction of literals that has to be evaluated.

Now, Δ is expressed by using the meta-predicates *old* and *new*. The definition of a revised inconsistency indicator is adjusted in order to cope with evaluations in the current and in the new database state. A revised inconsistency indicator $II(L, \mathcal{N})$ is expressed with the adjusted Δ by:

$$\Delta^{L}_{\mathcal{N}}, new(II_L).$$

For instance, $\Delta^{a(X)}_{\neg d(V,Y)}$ of EXAMPLE 5.2 is equal to

$$new(b(X, Y)), old((e(Z), \neg b(V, Y))), new(\neg c(Y, Z)).$$

So, the revised inconsistency indicator $II_1(a(X), \neg d(V, Y))$ can be formulated as follows:

$$\Delta^{a(X)}_{\neg d(V,Y)}, new(\neg d(X, X)) =$$

$$new(b(X, Y)), old((e(Z), \neg b(V, Y))), new(\neg c(Y, Z)), new(\neg d(X, X)).$$

The related revised inconsistency rule is therefore:

$$IR : inconsistent(del(d(V, Y))) \Leftarrow$$

$$new(b(X, Y)), old((e(Z), \neg b(V, Y))), new(\neg c(Y, Z)), new(\neg d(X, X)).$$

Further, predicates *del* and *ins* are used to state that an update is deleted or inserted to the database. Suppose we have a deletion of the fact $d(1, 2)$, then the application of IR implies the evaluation of the body of the rule to which the substitution $\{V/1, Y/2\}$ is applied and $T = \{del(d(1, 2))\}$. This expression gives a more accurate characterization of the check needed for a proof of consistency than the expression in the previous version of this method based on inconsistency rules. In the previous version, we had to check:

$$new((a(1), \neg d(1, 1))).$$

REMARK There may be a difference in the characterization of the necessary check, but this does not mean that the performance between both characterizations differs. For instance, evaluating in both methods $\neg d(1, 1)$ first could lead to an early failure of both queries, when $d(1, 1)$ is derivable in the updated database. However, the decision to evaluate $d(1, 1)$ first is the responsibility of the query evaluator. On the other hand, when $\neg d(1, 1)$ holds in the updated database, the evaluation of $a(1)$ implies a search through rule R_1 and R_2, while in the revised inconsistency rule only the rule R_1 has to be applied.

The responsibility of the order of evaluation of the subgoals is a responsibility of the query optimizer. However, *FICCS* can deliver the inconsistency rules in an optimized form, because when fully instantiating the head of an inconsistency rule, body literals may also be partially or even fully instantiated; so, we can determine the order of body literals by using this knowledge. Note, however, the subtle difference between check optimization and query optimization. Also note that in the case of negation inconsistency trees and inconsistency rules are derived in the same way as in the case without negation.

5.1.2.2 Existential Quantifiers in the Method based on Inconsistency Rules

Formulas for expressing constraints can be extended. We introduce existential quantifiers in our language. The restricted quantified formulas of Bry, Decker and Manthey, see [BDM87], are used.

DEFINITION 5.3 A closed first-order formula is *restricted quantified* if it has one of the following forms:

$$\exists X_1 \exists X_2 \ldots \exists X_m [A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge Q],$$
$$\forall X_1 \forall X_2 \ldots \forall X_m [A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow Q],$$

where $A_1, A_2, \ldots, A_n$ are atoms such that each variable X_i appears in an atom A_j, and where Q is $true$ or $false$, some quantifier free formula for which each variable must appear in the sequence $X_1, X_2, \ldots, X_m$, or some formula of the form above in which the variables $X_1, X_2, \ldots, X_m$ are not bound.

Restricted quantified formulas are domain independent, because negative literals are pushed down in the formula such that variables in those literals appear in positive atoms in a higher level of the formula. Hence, the evaluation of negative literals only takes place when they are fully instantiated by the atoms containing those variables.

In order to be applicable to the method based on inconsistency rules, restricted quantified formulas are transformed into generalized inconsistency indicators. Such formulas are composed by using the following expressions:

$$\neg(\exists X_1 \exists X_2 \ldots \exists X_m [A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge Q]),$$

$$\exists X_1 \exists X_2 \ldots \exists X_m [A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge Q],$$

for which the conditions for $A_1, A_2, \ldots, A_n$ and Q in DEFINITION 5.3 hold.

EXAMPLE 5.3 Let D be a deductive database. Let

$$II : \exists X \exists Y [a(X, Y), b(Y), \neg(\exists Z_1 \exists Z_2 [c(X, Z_1), d(Z_2, X)])].$$

Let a, b, c and d all be base relations. Suppose D is consistent with respect to II. As before we look at possible updates of each of the conjuncts, i.e., $a(X, Y)$, $b(Y)$ and $\neg(\exists Z_1 \exists Z_2 [c(X, Z_1), d(Z_2, X)])$ of the extended inconsistency indicator. The first two are handled like before; potential update AND/OR trees are drawn for each of these literals. It is the third expression that may

hold after an update which could make the whole inconsistency indicator true. So, we must find out for which kind of updates the negation of this expression, i. e., $\exists Z_1 \exists Z_2[c(X, Z_1), d(Z_2, X)]$ could be false, in the updated database. It is easy to see that such updates are deletions of facts in c or d. For instance, let $del(d(1, 2))$ be a deletion, then $\exists Z_1 \exists Z_2[c(2, Z_1), d(Z_2, 2)]$ may become false because $d(1, 2)$ was the only instance of $d(Z_2, 2)$ in D. However, it must be checked if this had been the case. Therefore, after update $del(d(1, 2))$ the instantiated inconsistency indicator that has to be checked is:

$$\exists Y[a(2, Y), b(Y), \neg(\exists Z_1 \exists Z_2[c(2, Z_1), d(Z_2, 2)])].$$

Note that variables that are bound on a lower level must not be instantiated by the update in order to reach the proper check. So, Z_2 is not instantiated and therefore $d(Z_2, X)$ cannot be left out of the expression, because we must check if some other instance of $d(Z_2, X)$ besides for $Z_2 = 1$ still exists. The inconsistency rule for a deletion in d is therefore:

$$inconsistent(del(d(V, X)) \Leftarrow new((a(X, Y), b(Y), \neg(c(X, Z_1), d(Z_2, X)))).$$

Suppose that d is a derived predicate defined as follows:

$$d(X, Y) \longleftarrow e(X, Z), f(Z, Y).$$

Now, a deletion of a fact in d may be induced by either a deletion in e or f. The inconsistency rules with respect to induced deletions in d are therefore:

$$inconsistent(del(e(V, W)) \Leftarrow new((a(X, Y), b(Y), \neg(c(X, Z_1), d(Z_2, X))))$$

and

$$inconsistent(del(f(V, X)) \Leftarrow new((a(X, Y), b(Y), \neg(c(X, Z_1), d(Z_2, X)))).$$

Note that a deletion in e requires a complete check of the inconsistency indicator. When a nesting of negations is involved only variables that appear in the highest level of the formula are instantiated, while the predicates in a level below the top level are not replaced by any expression, but may be instantiated. In other words the update expression of such a predicate is (a possibly instantiated version of) the predicate itself.

When a deductive database system is extended to a general theorem prover, it is possible to reason with general formulas with universal as well as existential quantifiers. When such formulas can be evaluated, inconsistency rules can contain those formulas in their body. In the literature several theorem provers can be found. In [SO93] and [Oph92] instead of resolution a tableaux approach is used. The theorem prover presented in [Oph92] is able to evaluate most of the general formulas appearing in the body of the inconsistency rules efficiently. So, theorem provers are interesting when considering integrity constraint checking, because sometimes one has to formulate integrity constraints in a more general language.

5.1.3 Extended Update Expressions

The update expression in the body of a revised inconsistency rule is some condition for which the remainder of the inconsistency indicator must be evaluated. In other words, when an instance of the update expression is found, the remainder of the inconsistency indicator is evaluated for this instance. The update expression expresses the derivation of induced updates, while we could have defined the update expression such that it expresses only the derivation of effective induced updates, because only effective induced updates are needed for checking the consistency of a database. As a consequence, the *ineffectiveness check* has to be incorporated into the update expression.

EXAMPLE 5.4 Consider the update expression $\Delta^{a(X)}_{\neg d(V,Y)}$ of EXAMPLE 5.2. Let $\blacktriangle^{a(X)}_{\neg d(V,Y)}$ denote the update expression containing an ineffectiveness check. This means that we have to check if an induced update in $a(X)$ is effective. In other words, we have to be sure that when $\Delta^{a(X)}_{\neg d(V,Y)}$ holds for some instance of X also $\neg old(a(X))$ is true, before the remainder of the revised inconsistency indicator, i.e., $new(\neg d(X, X))$, is evaluated. Now, the revised inconsistency indicator $II_1(a(X), \neg d(V, Y))$ is formulated as:

$$\blacktriangle^{a(X)}_{\neg d(V,Y)}, new(\neg d(X, X))$$

which is equal to

$$\Delta^{a(X)}_{\neg d(V,Y)}, \neg old(a(X)), new(\neg d(X, X)).$$

The ineffectiveness check in the update expression could prevent the check of the inconsistency indicator in the case of a relevant ineffective induced update at the cost of the ineffectiveness check for that induced update. The choice between those options depends on the interaction between fact and rule base and the update.

Instead of $\Delta^{L}_{\mathcal{N}}$, we could define a more advanced definition of update expression by using estimations about the probabilities of deriving ineffective induced updates depending on L and $\mathcal{N}$. The update expression can be determined at run-time including the decision if the ineffectiveness check is desirable or not, or at compile time when relevant information is stored for earlier sessions in which such a decision is made.

5.2 Extended Transactions

Instead of only permitting facts in transactions, we could also permit updates of rules or constraints in transactions. In this section, the adjustments for the method of integrity checking based on inconsistency rules when rules and constraints are permitted in the transactions are elaborated.

5.2.1 Rules in Transactions

Besides updating the fact base the rule base could also be updated. However, when a rule is deleted a lot of new facts may not be derivable anymore and when a rule is inserted a lot of facts may be derivable.

DEFINITION 5.4 Let $ins(R)$ be an insertion of a deductive rule in a transaction T to a database D, where $R : C \leftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$. Let $C' = C\sigma$, where σ is a substitution for which $(L_1 \wedge \cdots \wedge L_n)\sigma$ is true in D_T. Then C' is called *positively directly induced by* R over D_T.

In case of a deletion of a rule a similar definition is used.

DEFINITION 5.5 Let $del(R)$ be a deletion of a deductive rule in a transaction T to a database D, where $R : C \leftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$. Let $C' = C\sigma$, where σ is a substitution for which $(L_1 \wedge \cdots \wedge L_n)\sigma$ is true in D. Then $\neg C'$ is called *negatively directly induced by* R over D_T.

DEFINITION 5.6 A literal L is *directly induced by* R if L is either positively or negatively directly induced by R over D_T.

When it is clear from the context which transaction T with respect to which database D is meant, we skip the phrase "over D_T".

REMARK Because we assumed that rules are range-restricted, C' in DEFINITION 5.4 and DEFINITION 5.5 is ground.

DEFINITION 5.7 Let T be a transaction to a database D and R a deductive rule appearing in T. A literal is *induced by* R iff

(i) it is directly induced by R, or

(ii) it is induced by a literal induced by R.

Like in the case of fact updates, rule updates can imply induced updates.

DEFINITION 5.8 Let T be a transaction to a database D in which a rule R appears. Each literal induced by R will be called an *induced update* with respect to transaction T (or simply *induced update*if it is clear from the context which transaction is involved).

Similar definitions are defined for the potential case.

DEFINITION 5.9 Let $ins(R)$ be an insertion of a deductive rule in a transaction T to a database D, where $R : C \leftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$. Then we say that C *positively directly depends on* R.

DEFINITION 5.10 Let $del(R)$ be a deletion of a deductive rule in a transaction T to a database D, where $R : C \leftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$. Then $\neg C$ *negatively directly depends on* R.

DEFINITION 5.11 A literal L *directly depends on* a rule $R : C \leftarrow L_1 \wedge L_2 \wedge \cdots \wedge L_n$ if L either positively or negatively directly depends on R.

DEFINITION 5.12 Let T be a transaction to a database D and R a deductive rule appearing in T. A literal *depends on* R iff

(i) it directly depends on R, or

(ii) it directly depends on a literal depending on R.

Like in the case of fact updates, rule updates can imply potential updates.

DEFINITION 5.13 Let T be a transaction to a database D in which a rule R appears. Each literal depending on R is called *a potential update with respect to transaction* T (or simply *potential update* if it is clear from the context which transaction is involved).

DEFINITION 5.14 Let T be a transaction to a database D in which a rule R appears. We call R *relevant* for the integrity check of the updated database D_T, or just *relevant* when it is clear which D and T is meant, if there exists a literal depending on R that is relevant to some inconsistency indicator, else it is called *irrelevant*.

REMARK In DEFINITION 5.7 (resp. DEFINITION 5.12) we use the notion of "induced by a literal" (resp. "depends on a literal") from DEFINITION 2.6 (resp. DEFINITION 2.15). The sign of the induced update (resp. potential update) will depend on the sign of the rule in the transaction, i. e., positive in the case of insertion and negative in the case of deletion.

Note that previous definitions in which induced updates are used, for instance in DEFINITION 2.11 and DEFINITION 2.21, resulting from an update in the transaction, can still be used if we interpret an update as a fact update or a rule update.

As in the case of fact updates, we are searching for the relevant induced updates induced by the rule updates in a transaction. However, those induced updates do not depend on an update of any base relation, but are only caused by updates of a derived relation. So, in the case of rule updates an intermediate node in the potential update AND/OR tree may be updated without having updated any of its child nodes. This means that here all nodes in potential update AND/OR trees are considered as updatable nodes, since only the leaf nodes are updatable by fact updates, while all other nodes are updatable by rule updates.

EXAMPLE 5.5 Let D be a database consisting of the following rule base and fact base:

RULES

R_1: $c(X, Y) \longleftarrow a(X, Z), b(Z, Y)$

R_2: $d(X) \longleftarrow c(X, X)$

FACTS

F_1: $a(1, 10)$

F_2: $b(10, 10)$

From this database the fact $c(1, 10)$ is derivable. However, in D no fact in d is derivable. This situation changes when the rule $R : d(X) \longleftarrow a(X, Y), c(X, Y)$ is inserted. In the updated database $d(1)$ holds, i. e., $d(1)$ is positively directly induced by R. So, R produces $d(1)$ as an induced update. Because d can be updated independently of updates in a or b, d must be considered as updatable.

Let $R : C \longleftarrow L_1 \wedge \cdots \wedge L_n$ be a rule in a transaction T to a database D. Let L be a database literal that appears in an inconsistency indicator. Suppose that C appears as a node in the potential update AND/OR tree T_L. When $ins(R)$ (resp. $del(R)$) is present in T, then for each instance of $L_1 \wedge L_2 \wedge \cdots \wedge L_n$ that holds in D_T (resp. D) an induced insertion (resp. deletion) is derived. However, the update $ins(R)$ (resp. $del(R)$) does not update the relations appearing in the body of R. Only the derived relation of C is updated. The updates in C can also lead to induced updates described by DEFINITION 5.7 and DEFINITION 5.8.

Like in the case of fact updates these induced updates can affect the integrity of the database. So, there is no reason for treating C differently compared to any leaf node $\mathcal{N}$ in T_L. The only difference is the way these updates to nodes are derived. In the case of $\mathcal{N}$ the updates must appear in a transaction, while in the case of C the updates must be instances of the head literal of a rule R obtained by evaluating the body of R. So, in the case of an insertion of R we have to apply all generated revised inconsistency rules for each update in C, which is determined by true instances of $L_1 \wedge L_2 \wedge \cdots \wedge L_n$ in D_T.

We can generate a revised inconsistency indicator $II(L, C)$ with respect to L and C, where L in II is replaced by Δ_C^L and the remainder is evaluated in the updated database, i. e., $II(L, C) = \Delta_C^L, new(II_L)$. This leads to the inconsistency rule

$$inconsistent(ins(C)) \Leftarrow \Delta_C^L, new(II_L)$$

which is only applied when an insertion in C appears.

In the same manner, we can generate a revised inconsistency indicator $II(L, \neg C)$ with respect to L and $\neg C$, where L in II is replaced by $\Delta_{\neg C}^L$ and the remainder is evaluated in the updated database. This will lead to the inconsistency rule

$$inconsistent(del(C)) \Leftarrow \Delta_{\neg C}^L, new(II_L).$$

which is only applied when a deletion in C appears.

REMARK All rule updates with a head literal C are handled identically, namely, for each rule its body has to be evaluated first in order to find the induced updates in C to which the revised inconsistency rules have to be applied.

EXAMPLE 5.6 Consider the database of EXAMPLE 5.5. Suppose D must obey the inconsistency indicator $II : d(X), X < 0$. Let $T = \{ins(b(10, 1)), ins(c(X, X) \longleftarrow b(X, X))\}$ be a transaction to D. The transaction causes new derivable facts in c, namely, $c(10, 10)$ by the rule in the transaction and fact F_2 and $c(1, 1)$ by rule R_1 and the fact in the transaction. Both induced updates will influence II, although they influence the II in different ways. Note that $c(1, 1)$ is induced by $b(10, 1)$ and the corresponding check is performed by the revised inconsistency rule

$$inconsistent(ins(b(Z, X)) \Leftarrow new(a(X, Z)), new(X < 0).$$

The corresponding check for the induced updates in c, caused by the rule update, is performed by the revised inconsistency rule

$$inconsistent(ins(c(X, X)) \Leftarrow new(X < 0).$$

Note that this revised inconsistency rule is only applied for $c(10, 10)$.

As pointed out in the remark above, revised inconsistency rules for C are in a sense alike. We can unite these revised inconsistency rules into some generic revised inconsistency rule. For instance, let there be an insertion of a rule $R : C \longleftarrow B$, where B represents the conjunction of body literals of R. We are now reconstructing the revised inconsistency rules for rules in such a way that the update of a rule directly triggers the related revised inconsistency rule. This can be done by using a general format of the rule in the head of the inconsistency rule and by incorporating the test for finding the induced updates in C, i. e. , the evaluation of the body of the rule, into the body of the revised inconsistency rule. In our example, the following revised inconsistency rule is stated:

$$inconsistent(ins(C \longleftarrow B)) \Leftarrow new(B), \Delta_C^L, new(II_L).$$

Similarly, the corresponding revised inconsistency rule for the deletion of R is

$$inconsistent(del(C \longleftarrow B)) \Leftarrow old(B), \Delta_{\neg C}^L, new(II_L).$$

Although we do not know exactly the rule that is involved, it is possible to construct those inconsistency rules at compile time. In order to be able to construct Δ_C^L or $\Delta_{\neg C}^L$ the literal C must be known at compile time. However, the literals of B are only needed at run-time, because they only have to be evaluated by the query evaluator. Hence, we only have to contruct inconsistency rules for literals that appear as nodes in the potential update AND/OR tree. All other literals will not affect any constraint and therefore updates of rules for which the head literal is one of those literals can be done immediately, i. e. , without an inconsistency check. Further, when we assume that the database is kept structured and an extensional predicate cannot be made intensional by inserting a rule defining an extensional predicate, then the only nodes in the potential update AND/OR tree which are relevant with respect to rule updates are the nodes corresponding to derived relations. So, general revised inconsistency rules are constructed in which B is not specified at compile time, but is instantiated at run-time.

Note that when a rule update appears in the transaction it can directly be applied to an inconsistency rule. This is illustrated by the next example.

EXAMPLE 5.7 Consider the situation in EXAMPLE 5.6. Now, the following revised inconsistency rule is used for checking rule updates for which its head matches with $c(X, Y)$:

$$inconsistent(ins(c(X, Y) \longleftarrow B)) \Leftarrow new(B), new(X < 0).$$

So, when $ins(c(X, X) \longleftarrow b(X, X))$ appears in the transaction, it matches the head of this revised inconsistency rule. Further, the meta-variable B is bound to $b(X, X)$ and $new(b(X, X))$ is evaluated first to find those instances of $c(X, X)$, i. e. , the induced updates caused by the insertion of the rule, for which the remainder of the body of the revised inconsistency rule, i. e. , the revised inconsistency indicator, is evaluated.

DEFINITION 5.15 An inconsistency rule (or a revised inconsistency rule) which is defined for a fact update is called a *base inconsistency rule*. An inconsistency rule (or a revised inconsistency rule) which is defined for a rule update is called a *derived inconsistency rule*.

Note that when a rule update appears in the transaction then the set of base and derived inconsistency rules must also be updated. Two cases may appear. First, the rule is not relevant to any inconsistency indicator, in which case the set of inconsistency rules is not changed. Second, the rule is relevant to an inconsistency indicator, in which case the corresponding derived inconsistency rule is applied.

REMARK A rule update may have effect on one or more potential update AND/OR trees and therefore it has effect on the set of inconsistency rules. Therefore, the set of inconsistency rules has to be updated, but the adjustment of the set of inconsistency rules must be rolled back when the transaction leads to an inconsistency. Note that the set of inconsistency rules does not have to be constructed from scratch, but can be built incrementally. We only need to construct new base inconsistency rules, when a new leaf node happens to appear in some potential update AND/OR tree. Further, when a leaf node is deleted we only have to delete those inconsistency rules that were built from this leaf node. Also when derived predicates are introduced, then the related derived inconsistency rules also have to be constructed. Other inconsistency rules that were not influenced by the transaction do not have to be computed again.

5.2.2 Constraints in Transactions

The definition of update can be extended to allow constraints as well.

When a database D satisfies a set of constraints, then D will satisfy each subset of constraints as well. So, when a constraint is deleted the consistency of the database is not affected in any way. However, some administration has to be performed, i. e. , keeping track of the deletions of inconsistency rules that corresponds to the deletion of that constraint. In the case of inconsistency rules, this task is an easy one. The body of an inconsistency rule is derived from a constraint. When an identifier for a constraint is added to the head of the inconsistency rule, we can recognize inconsistency rules corresponding to the constraint easily. So, the deletion of such inconsistency rules can be done instantly.

When a constraint is inserted to the database not only the induced updates are needed to check the constraint, but also the fact base as a whole. Here our assumption that the database is consistent with respect to the specified constraints, including the inserted one, may fail, because the constraint may be violated in the current database state. It is inaccurate to reject a constraint only based on the fact that it is violated in D, because we only demand that it holds in the updated database. For, it is possible that there are certain updates in the transaction that guarantee that the constraint is satisfied in the updated database. However, when it is proved that a constraint holds in D, then we could handle the constraint as any other constraint, i. e. , they only have to be

evaluated on the changed part of the database. The necessary check of constraints that appear in the transaction can be represented by:

$$inconsistent(ins(ii(II))) \Leftarrow new(II),$$

where $ii(II)$ is an expression that represents any inconsistency indicator and where II is a variable representing a conjunction of literals. Note that deletions of constraints require no inconsistency rules. So, the necessary checks for constraints in transactions are representable as one generic inconsistency rule.

Further note that for each new constraint in the transaction other inconsistency rules must be derived, namely those corresponding to fact updates that influence this new constraint. This means that for each new constraint revised inconsistency rules must be generated from scratch. However, when the constraint contains a literal for which a potential update AND/OR tree already exists and the potential update AND/OR tree is somehow stored in the system, this knowledge can be reused in order to build these revised inconsistency rules.

REMARK In this thesis the insertion of a constraint violating the updated database state is forbidden, because we supposed a signaling approach instead of a maintenance approach. However, in real database applications one may want to insert the constraint at any cost. This could be done by searching for those facts (or even rules) which may be responsible for this inconsistency and reconsider the truth of those facts.

5.2.3 Replacements in Transactions

Inconsistency rules are natural constructs to handle the consistency of a database efficiently when considering replacements of facts. Suppose we have a modification of a fact $a(c_1, c_2, \ldots, c_n)$ in which the constant c_j is replaced by c_j'. We express such a modification by the expression:

$$rpl(a(c_1, c_2, \ldots, c_{j-1}, c_j \rightarrow c_j', c_{j+1}, \ldots, c_n)).$$

A replacement is often seen as an insertion after a deletion. In this section, we will argue why it is not wise from an integrity checking point of view to divide a replacement into an insertion and a deletion, before the integrity of the database is checked. Some relevant information hidden in the replacement and relevant for the integrity check will be lost. For instance, the replacement above can be divided in:

$$del(a(c_1, c_2, \ldots, c_{j-1}, c_j, c_{j+1}, \ldots, c_n))$$

and

$$ins(a(c_1, c_2, \ldots, c_{j-1}, c_j', c_{j+1}, \ldots, c_n)).$$

Each of these updates can be applied to revised inconsistency rules as if they were two independent updates in the transaction. Note that we have to check whether removing

$$a(c_1, c_2, \ldots, c_{j-1}, c_j, c_{j+1}, \ldots, c_n)$$

does not influence the consistency of the database and whether the appearence of the fact

$$a(c_1, c_2, \ldots, c_{j-1}, c'_j, c_{j+1}, \ldots, c_n)$$

does not influence the consistency of the database either. However, this division of the replacement may lead to a considerable overhead in the check.

First, consider the insertion of $a(c_1, c_2, \ldots, c_{j-1}, c'_j, c_{j+1}, \ldots, c_n)$. Suppose some inconsistency rules exist applicable to the insertion, i. e. , inconsistency rules for which the head is

$$\mathit{inconsistent}(\mathit{ins}(a(X_1, X_2, \ldots, X_n))).$$

Note that the deleted fact $a(c_1, c_2, \ldots, c_{j-1}, c_j, c_{j+1}, \ldots, c_n)$ was present in the old database state and therefore evaluating $\mathit{inconsistent}(\mathit{ins}(a(c_1, c_2, \ldots, c_{j-1}, c_j, c_{j+1}, \ldots, c_n)))$ must be false, resulting in an evaluation in the old database state.

For each inconsistency rule for which X_j is not an update variable nothing should be checked, because the body of those inconsistency rules is instantiated by

$$a(c_1, c_2, \ldots, c_{j-1}, c'_j, c_{j+1}, \ldots, c_n)$$

which is instantiated exactly the same by $a(c_1, c_2, \ldots, c_n)$. So, the choice of either of these two facts does not make any difference in the evaluation of the body of the revised inconsistency rule. Two cases can be considered:

(i) The transaction only contains the replacement.

(ii) The transaction contains other updates besides the replacement.

In the first case, the body of the revised inconsistency rule does not hold in the updated database, because the evaluation of this body is an exact copy of the corresponding evaluation of this body for the deleted fact in the old database, which does not hold because the database is consistent before the update. In the second case, other updates may influence the evaluation of the body of the inconsistency rule but their influence is performed through other inconsistency rules. For, the truth of the body of the inconsistency rule for the replacement in the update database is not different when we had used the replacent in the evaluation. This means that the violation of the corresponding inconsistency indicator is not caused by the replacement but by some other element of the transaction. So, some other revised inconsistency rule applicable to some other update in the transaction is responsible for checking the possible change in the integrity of the database. Checking the consistency by applying an inconsistency rule, for which X_j is not an update variable, to our replacement would therefore be redundant.

However, if X_j is an update variable, then we have to check the body of the inconsistency rule with respect to the insertion of $a(c_1, c_2, \ldots, c_{j-1}, c'_j, c_{j+1}, \ldots, c_n)$, because the check depends on a changed value in a fact of a.

Because the fact $a(c_1, c_2, \ldots, c_n)$ no longer exists, we also have to check the body of each inconsistency rule concerning deletions, for which their heads look like

$$inconsistent(del(a(X_1, X_2, \ldots, X_n)))$$

and where X_j is an update variable.

REMARK A replacement is only divided into a deletion and an insertion for revised inconsistency rules for which it is relevant, i. e. , the replacement is in arguments that are relevant for the check. Revised inconsistency rules not relevant to the replacement do not have to be applied.

These considerations show that in case of replacements the revised inconsistency rules with respect to insertions and deletions can be used for this purpose by using some meta-knowledge of the revised inconsistency rules, namely, the knowledge of the place in which update variables appear in the revised inconsistency rules. When the revised inconsistency rules are constructed, this knowledge can be made explicit and is put in the specification of the revised inconsistency rules. This is done by logic programming techniques, where the numbers of the arguments containing update variables are collected in a list, which we will call the *update variable list*. For instance, the head of an inconsistency rule could look like

$$inconsistent([1, n], del(a(X_1, X_2, \ldots, X_n)))$$

which means that the first and the n-th argument in $a(X_1, X_2, \ldots, X_n)$ will also appear in the body of that rule. When some replacement takes place we can also make a list that tells us in which positions of the relation the modifications take place. We call this list the *replacement list*. When the intersection of the update variables list and the replacement list is empty, the corresponding inconsistency rule does not have to be executed. So, in case of

$$rpl(a(c_1, c_2, \ldots, c_{j-1}, c_j \rightarrow c_j', c_{j+1}, \ldots, c_n))$$

for which the replacement list only contains j, this inconsistency rule does only have to be executed when $j = 1$ and $j = n$. Note that in those cases the inconsistency rule must be applied to the fact before it was replaced, i. e. , to $a(c_1, c_2, \ldots, c_n)$.

When the head of the inconsistency rule had been

$$inconsistent([1, n], ins(a(X_1, X_2, \ldots, X_n)))$$

then it should be applied to the replacent of $a(c_1, c_2, \ldots, c_n)$, i. e. , to

$$a(c_1, c_2, \ldots, c_{j-1}, c_j', c_{j+1}, \ldots, c_n)).$$

In order to clarify replacements a more practical example is given.

EXAMPLE 5.8 Suppose a database D contains the three relations called `citizen`, containing a person's id in its first argument, the person's spouse in its third argument and the person's sex

in its last argument and other information with respect to the person's address in the other arguments which are not specified because they are not important, `occupation`, containing information about a person's occupation, a person's id, a person's first and last name, date of application, department name, room number and job name, and `taxes`, containing information about the taxes a person is indebted, given in the fifth argument of this predicate. The following rules are present in D:

R_1: $has\_job(X, Z) \longleftarrow \texttt{occupation}(X, \_, \_, \_, \_, \_, Z), \neg Z = student$

R_2: $tax\_type(X, T) \longleftarrow \texttt{taxes}(X, \_, \_, \_, N, \_), has\_job(X, Z), taxes\_to\_type(N, T).$

In these rules only the distinguished variables and the connection variables are presented, the other variables are kept unspecified. Suppose the following inconsistency indicator is specified for D:

II_1: $tax\_type(X, T), \texttt{citizen}(X, \_, Y, \_, \_, \_, \_, male), tax\_type(Y, T1), \neg T = T1.$

The inconsistency indicator corresponds to the constraint which expresses that when two persons are married and they have both a tax type it must be the same tax type. Note that *taxes_to_type* is an evaluable predicate, computing the type of the tax based on its rate. The revised inconsistency rules belonging to this part of the database are:

$inconsistent([1, 3], ins(\texttt{citizen}(X, \_, Y, \_, \_, \_, \_, male)) \Leftarrow$

$tax\_type(X, T), tax\_type(Y, T1), \neg T = T1$

$inconsistent([1, 7], ins(\texttt{occupation}(X, \_, \_, \_, \_, \_, Z)) \Leftarrow$

$\neg Z = student, \texttt{citizen}(X, \_, Y, \_, \_, \_, \_, male), \texttt{taxes}(X, \_, \_, \_, N, \_),$

$taxes\_to\_type(N, T), tax\_type(Y, T1), \neg T = T1$

$inconsistent([1, 7], ins(\texttt{occupation}(Y, \_, \_, \_, \_, \_, Z)) \Leftarrow$

$\neg Z = student, \texttt{citizen}(X, \_, Y, \_, \_, \_, \_, male), \texttt{taxes}(Y, \_, \_, \_, N1, \_),$

$taxes\_to\_type(N1, T1), tax\_type(X, T), \neg T1 = T$

$inconsistent([1, 5], ins(\texttt{taxes}(X, \_, \_, \_, N, \_)) \Leftarrow$

$has\_job(X, Z), \texttt{citizen}(X, \_, Y, \_, \_, \_, \_, male), tax\_type(Y, T1),$

$taxes\_to\_type(N, T), \neg T = T1$

$inconsistent([1, 5], ins(\texttt{taxes}(Y, \_, \_, \_, N1, \_)) \Leftarrow$

$has\_job(Y, Z), \texttt{citizen}(X, \_, Y, \_, \_, \_, \_, male), tax\_type(X, T),$

$taxes\_to\_type(N1, T1), \neg T1 = T.$

Suppose that D is updated by the following replacement:

$rpl(\texttt{occupation}(7323, John, Parker, 1/1/97 \rightarrow 1/6/97,$

$R\&D, P3.210, student \rightarrow researcher))$

which means that student John Parker, working at the company from the beginning of the year 1997, becomes a researcher at the first of june that year. Note that there are two revised inconsistency rules applicable to this replacement, i. e., the second and the third one above. These inconsistency rules only involve insertions, so, only the insertion part of the replacement may be applied to these inconsistency rules. In order to determine if those inconsistency rules have to be executed, the replacement list for this replacement has to be determined. The fourth and seventh argument of the `occupation` fact are changed, so, the replacement list is equal to [4, 7]. Because the intersection of [4, 7] and [1, 7] is not empty, the insertion

$$ins(\texttt{occupation}(7323, John, Parker, 1/6/97, R\&D, P3.210, researcher))$$

is applied to the second and third revised inconsistency rule. Consider the following replacement

$$rpl(\texttt{occupation}(7323, John, Parker, 1/1/97, R\&D \rightarrow IS,$$
$$P3.210 \rightarrow HG7.33, student)).$$

Because the related replacement list is equal to [5, 6] which is disjunct from the update variable list [1, 7] of the revised inconsistency rules with respect to insertions in `occupation`, no check has to be performed. An implementation of the automatic checking of replacements can be found in CHAPTER 6.

Note that some general modifications are handled as well by this approach. For instance, we can represent general replacements like

"replace in table a each c in column j by c'"

by $rpl(a(X_1, X_2, \ldots, X_{j-1}, c \rightarrow c', X_{j+1}, \ldots, X_n))$. As before, the intersection of the update variables list of some inconsistency rule and the replacement list for the replacement can determine if the inconsistency rule is checked for this general replacement. Note that when this general replacement corresponds to a great number of replacements, this can lead to an enormous gain of checking time, especially when comparing it to methods that handle a replacement as a deletion followed by an insertion.

5.3 Rules Extended with Recursion

We could allow recursive rules in our deductive database. When the query evaluator can handle recursive rules efficiently (see [Nau86, Nau89]), then there is no problem in evaluating queries containing recursive predicates. Especially, no problem exists when inconsistency indicators do not contain any recursive predicate, in which case the method based on inconsistency rules can be used without any adjustment. However, when recursion is introduced in an inconsistency indicator, the application of inconsistency rules could lead to serious problems. This section describes some of these problems and gives a solution for some types of recursion.

5.3.1 Recursion in Inconsistency Rules

In the case of recursion, the definitions in CHAPTER 4 that lead to the definition of inconsistency rules can also be used. In this section, we will take a closer look at these definitions from the viewpoint of recursion.

The last condition in DEFINITION 4.1 for potential update AND/OR trees guarantees the finite application of recursive rules in order to avoid infinite branches in these trees, because evidently deductive databases only contain a finite number of rules containing a finite number of arguments. This is illustrated by the next examples.

EXAMPLE 5.9 Let D be a deductive database with the following rules defining the recursive predicate a:

RULES

R_1: $a(X, Y) \longleftarrow b(X, Y)$

R_2: $a(X, Y) \longleftarrow b(X, Z), a(Z, Y)$

Suppose $a(X, Y)$ is the root literal of some potential update AND/OR tree $T_{a(X,Y)}$. The branches of $T_{a(X,Y)}$ with respect to R_2 consist of the branch leading to $b(X, Z)$ and the branch leading to $a(Z, Y)$. Now, by applying R_2 to $a(Z, Y)$ only one subnode $b(Z, Z_1)$ is derived. A subnode $a(Z_1, Y)$ differs only in the first argument from $a(Z, Y)$, but these arguments are variables that do not occur in the root literal. So, $a(Z_1, Y)$ is redundant (redundant in the sense that there is no difference in instantiating root literal $a(X, Y)$ if we instantiate either $a(Z, Y)$ or $a(Z_1, Y)$; in both cases an insertion only binds variable Y). In FIGURE 5.2 only the development of nodes for the recursive rule R_2 is shown. Now, without the last condition in DEFINITION 4.1 redundant branches would appear in the potential update AND/OR tree, leading to nodes pointed by the dotted arrows.

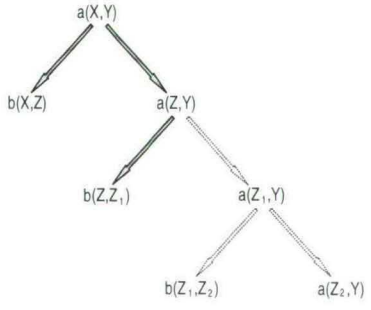


Figure 5.2: Redundancy in the Potential Update AND/OR Tree for EXAMPLE 5.9

Note that when $a(X, Y)$ appears in some inconsistency indicator, following the previous definitions for inconsistency rules, an insertion in relation b will lead to a full evaluation of $a(X, Y)$, because $b(Z, Z_1)$ does not have any variables in common with $a(X, Y)$. When the inconsistency rules for b are derived, we will derive an inconsistency rule of which the body is equal to the inconsistency indicator. So, the inconsistency indicator has to be fully checked. This is illustrated by using the next example, in which a and b are replaced by the predicates of *ancestor* and *parent*.

EXAMPLE 5.10 Consider the following Prolog definition of the *ancestor*-relation in terms of the *parent*-relation:

```
ancestor(X,Y) :-
  parent(X,Y).
ancestor(X,Y) :-
  parent(X,Z),
  ancestor(Z,Y).
```

When the predicate *ancestor* appears in an inconsistency indicator an update in the *parent*-relation will affect the *ancestor*-relation. Suppose that this inconsistency indicator is:

```
ancestor(X,Y), age_diff(X,Y,N), N < 15.
```

The inconsistency rules for this inconsistency indicator, which are generated to monitor the state of the database with respect to insertions of the *parent*-relation, are:

```
inconsistent(parent(X,Y)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
inconsistent(parent(X,Z)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
inconsistent(parent(Z,Y)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
inconsistent(parent(Z,Z1)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
```

Note that the last inconsistency rule subsumes all other rules, for this rule corresponds to a full check of the inconsistency rule. The reason for using the other rules is that one of them could lead to an earlier detection of some inconsistency than the most general one.

When a recursive predicate appears in an inconsistency indicator, an update in a nonrecursive relation which is used in the definition of the recursively defined relation may result in several induced updates in the recursive relation. From the perspective of efficient integrity checking it is not feasible to check the inconsistency indicator for the whole recursive relation. It suffices to check the inconsistency indicator for the effective induced updates only. Therefore, in the inconsistency indicator the recursive predicate should be replaced by an expression which exactly describes the change in the recursive relation. In general, the computation of changes in relations are explicitly

expressed by update expressions. However, the update expression as defined in DEFINITION 4.6 does not exactly describe the induced change of a recursively defined relation. But, in the case of recursively defined relations appearing in inconsistency indicators this general idea, namely, replacing the recursively defined predicate in the inconsistency indicator by an expression that comprises the change in that recursively defined relation, is still followed, although the potential update AND/OR tree is not sufficient for deriving this expression, as EXAMPLE 5.9 has shown. We illustrate the above in EXAMPLE 5.11.

When in example EXAMPLE 5.11 the base relation *parent* is updated, the update expression of *ancestor* with respect to *parent* must be an exact description of the change in *ancestor*. This update expression replaces the *ancestor* predicate in the inconsistency indicator. This will give a revised inconsistency indicator as before, leading in the same way to a revised inconsistency rule. So, all definitions can be used like before, except the definition of update expression. The proposed definition of an update expression in the case of recursion, will first be clarified by an example.

EXAMPLE 5.11 Consider the rules and the inconsistency indicator of EXAMPLE 5.10 and suppose the following *parent*-facts are involved.

```
parent(1,10).
parent(1,11).
parent(10,100).
parent(11,110).
parent(11,111).
parent(2,20).
parent(20,200).
parent(20,201).
```

The relations are presented as trees (see FIGURE 5.3). A person is a parent of someone if there is an arrow from that person to the other. A person is an ancestor of someone if there is a path from that person to the other. Suppose there is an insertion, say, $ins(parent(110, 2))$, to the parent-

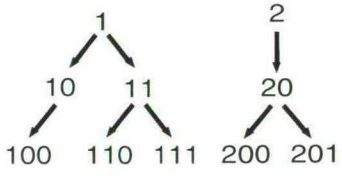


Figure 5.3: The parent relation for EXAMPLE 5.10

relation. What does this mean for the *ancestor*-relation? The *ancestor*-relation is updated by the insertion in the *parent*-relation too. This is represented in FIGURE 5.4. The two parent trees in FIGURE 5.4 are connected by the insertion. The insertion in this tree is depicted as an outlined

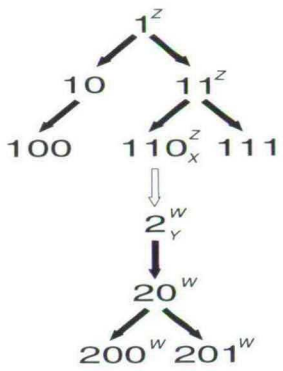


Figure 5.4: The updated parent relation

arrow. The starting node of the outlined arrow, representing the new *parent*-fact, is marked with X and the end node with Y. The insertion in the *ancestor*-relation can be described by the starting and end node of all paths from one node to another that have the outlined arrow in its path. In FIGURE 5.4 all possible starting nodes are marked with a Z, i. e., the node marked X or an ancestor of that node, and all possible end nodes with a W, i. e., the node marked Y or a node that has that node as an ancestor. Because all paths from a node marked with Z to a node marked with W go through the outlined arrow, all new ancestors can be computed for each insertion in the *parent*-relation by

```
ins(ancestor(Z,W)) :-
  ins(parent(X,Y)),
  (ancestor(Z,X) ; Z = X),
  (ancestor(Y,W) ; W = Y).
```

Now, evaluating *ins*(*ancestor*(Z, W)) gives all new ancestors implied by an insertion in the *parent*-relation.

We can restrict ourselves to the new instances of *ancestor*(X, Y) only. By using the results above the inconsistency rules mentioned in EXAMPLE 5.10 can be replaced by the single rule:

```
inconsistent(ins(parent(X,Y))) :-
  (ancestor(Z,X) ; Z = X),
  (ancestor(Y,W) ; W = Y),
  age_diff(Z,W,N), N < 15.
```

Note that *ancestor*(Z, W) is replaced by the expression

$$(\textit{ancestor}(Z, X)\ ;\ Z = X), (\textit{ancestor}(Y, W)\ ;\ W = Y)$$

where X and Y are variables which are instantiated by the insertion in parent. Therefore, the update expression with respect to *ancestor*(Z, W) and an insertion *parent*(X,Y) is defined as:

$$new((ancestor(Z, X)\ ;\ Z = X), (ancestor(Y, W)\ ;\ W = Y))$$

REMARK This is a considerable gain in efficiency compared to other existing methods that allow recursion. In case of methods based on potential updates the check of constraints with lineair recursive parts will lead to a full check of inconsistency indicators. In some cases the use of recursive relations is permitted in rules but avoided in the inconsistency indicators themselves. Hence, a full check of inconsistency indicators with recursion is avoided.

In this section, only one specific type of recursion is examined, namely a specific type of linear recursion. The next sections on recursion show the difficulties that appear when update expressions must be constructed for recursive relations for a more general type of recursion.

5.3.2 From Recursion to Linear Recursion

For reasons of efficiency, one specific type of recursion is interesting, namely, linear recursion. Because usually linear recursive queries can be handled more efficiently than nonlinear recursive queries, there is a need to transform nonlinear rules into linear ones. However, the question whether a nonlinear recursive rule can be transformed into a linear one or not is undecidable. So, a lot of effort is put in the search for decidable classes of nonlinear recursive rules. In the literature, some papers are dedicated to the conversion of recursive rules into linear recursive rules (see [Don92, GP95, IW89, Sar89, ZY87]), while some others are dedicated to the different types of linear recursive rules (see [LLH94, YKHH92]) and only a few are dedicated to linear recursion in deductive databases (see [HZL93]).

DEFINITION 5.16 A recursive rule R is called a *linear recursive rule* if there exists only one body literal of R for which its predicate is mutually recursive to the predicate of the head literal. Such a predicate is called a *linear recursive predicate*. Otherwise, a recursive rule is called a *nonlinear recursive rule*, in which case we say that the recursive predicate is a *nonlinear recursive predicate*.

Note that the predicate *ancestor* as defined in the previous section is linear recursive. Suppose we had defined the *ancestor* predicate by

```
ancestor(X,Y) :-
  parent(X,Y).
ancestor(X,Y) :-
  ancestor(X,Z),
  ancestor(Z,Y).
```

then we have created a nonlinear recursive predicate. However, it is obvious that both definitions describe the same relation *ancestor*. So, in this case the nonlinear version of *ancestor* is transformed into the linear one.

DEFINITION 5.17 A (nonlinear) recursive query is a query which contains a (nonlinear) recursive predicate. A linear recursive query is a recursive query which does not contain any nonlinear recursive predicates.

DEFINITION 5.18 A database D is called a *linear recursive database* if every recursive rule in D is linear recursive. Otherwise, D is called a *nonlinear recursive database*.

From now on databases are supposed to be linear recursive.

Not all linear recursive rules can be handled with the same ease. In order to study linear recursive rules they are first transformed into a normalized form. Consider the following general linear recursive definition in our database D:

$$R \longleftarrow L_1, L_2, \ldots, L_k$$

$$R' \longleftarrow L'_1, L'_2, \ldots, L'_m, R''$$

where R, R' and R'' are atoms containing a common predicate with the same arity. $L_1, L_2, \ldots, L_k$ and $L'_1, L'_2, \ldots, L'_m$ are literals.

REMARK R'' cannot be negative because of the stratification of D. Further, we suppose that there is no difference between $R' \longleftarrow R'', L'_1, L'_2, \ldots, L'_m$, the left linear recursive rule, and $R' \longleftarrow L'_1, L'_2, \ldots, L'_m, R''$, the right linear recursive rule, which means that we suppose a purely declarative interpretation of these rules. In Prolog the left linear recursive expression leads to indefinite bracktracking on R''. Therefore, a left linear recursive expression must first be transformed into a right linear one before being evaluated.

Because of the stratification each variable in $R = r(X_1, X_2, \ldots, X_n)$ will appear in a positive literal L_i. We define a new predicate a via a rule, which is not already known to D, for which $a(X_1, X_2, \ldots, X_n)$ is the head of the rule and $L_1 \wedge L_2 \wedge \cdots \wedge L_k$ is the body of the rule. Note that variables of $L_1, L_2, \ldots, L_k$ which do not appear in $X_1, X_2, \ldots, X_n$ are used locally in the body of the rule. Similarly, we define a new predicate b in order to replace $L'_1, L'_2, \ldots, L'_m$, which only contains variables appearing in R' and R''. By renaming the variables of R' such that it is equal to R and supposing that the predicate in R is r and has arity n, the rules above can be rewritten as:

R_1: $r(X_1, X_2, \ldots, X_n) \longleftarrow a(X_1, X_2, \ldots, X_n)$

R_2: $r(X_1, X_2, \ldots, X_n) \longleftarrow b(Y_1, Y_2, \ldots, Y_l), r(Z_1, Z_2, \ldots, Z_n)$

where $Z_1, Z_2, \ldots, Z_n$ are elements of $\{X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_l\}$. After presenting a linear recursive relation by rules R_1 and R_2, two types of linear recursion are considered, depending on the order of $Z_1, Z_2, \ldots, Z_n$ in $X_1, X_2, \ldots, X_n$ and $Y_1, Y_2, \ldots, Y_l$.

For instance, consider the following linear recursive definitions:

R_1: $r(X, Y) \longleftarrow a(X, Y)$

$$R_2\colon\ r(X, Y) \longleftarrow b(X, Z), r(Z, Y)$$
$$R_3\colon\ s(X, Y) \longleftarrow a(X, Y)$$
$$R_4\colon\ s(X, Y) \longleftarrow b(Y, Z), s(Z, X)$$

in which a and b are supposed to be base relations. Note that the only difference between r and s is the switch of the variables X and Y in the body of the recursive definition of r and s. This leads to update expressions for r and s with respect to a and b, which differ from each other significantly. Note that the recursively defined relations r and s are a generalization of the transitive closure of a relation, because it uses two possibly different base relations in its definition. This also means that an update of the recursively defined relations can be caused by either an update in a or b. When unfolding $r(X, Y)$ by applying rule R_1 after applying rule R_2 k times for $k = 0, 1, 2, \ldots$, then we find the following sequence of computations:

$$a(X, Y)$$
$$b(X, Z), a(Z, Y)$$
$$b(X, Z'), b(Z', Z), a(Z, Y)$$
$$b(X, Z''), b(Z'', Z'), b(Z', Z), a(Z, Y)$$
$$b(X, Z'''), b(Z''', Z''), b(Z'', Z'), b(Z', Z), a(Z, Y)$$
$$\vdots$$

and when unfolding $s(X, Y)$ by applying rule R_3 after applying rule R_4 k times for $k = 0, 1, 2, \ldots$, then we find the following sequence of computations:

$$a(X, Y)$$
$$b(Y, Z), a(Z, X)$$
$$b(Y, Z), b(X, Z'), a(Z', Z)$$
$$b(Y, Z), b(X, Z'), b(Z, Z''), a(Z'', Z')$$
$$b(Y, Z), b(X, Z'), b(Z, Z''), b(Z', Z'''), a(Z''', Z'')$$
$$\vdots$$

When looking at these unfolded definitions of r and s it appears that the switch in the definition of the variables X and Y has a great impact in the way the facts in a and b are chained. Suppose we have some facts with respect to a and b.

In case of r, the facts in r are derived by looking for chains of facts in a and b, where a fact in a starts a chain and is followed by a fact in b as long as the first argument of the current chain member is equal to the second argument of the previous chain member.

In case of s completely different chains are derived, which are centered around a fact in a, which

depends on the length of the chain. Consider a fact in s that is reached by k times applying rule R_4; then a chain of length $k+1$ of a fact in a and facts in b is derived. When k is odd, facts in b are chained $(k-1)/2$ times before a fact in a is reached, after which $(k+1)/2$ facts in b must be chained; when k is even, facts in b are chained $k/2$ times before a fact in a is reached, after which $k/2$ facts in b must be chained. For s the chaining process is originated by looking for correspondences between the second argument of the current chain member to the first argument of the previous chain member before reaching the fact in a by a correspondence in the first argument of a, and the first argument of the current chain member to the second argument of the previous chain member after reaching the fact in a. These two cases are illustrated in FIGURE 5.5 (where $Z^0 = Z$, $Z^{-1} = X$ and $Z^{-2} = Y$). So, each chain of facts in a and b of the format represented by the figure gives a fact in the recursively defined relation r and s respectively. This figure also

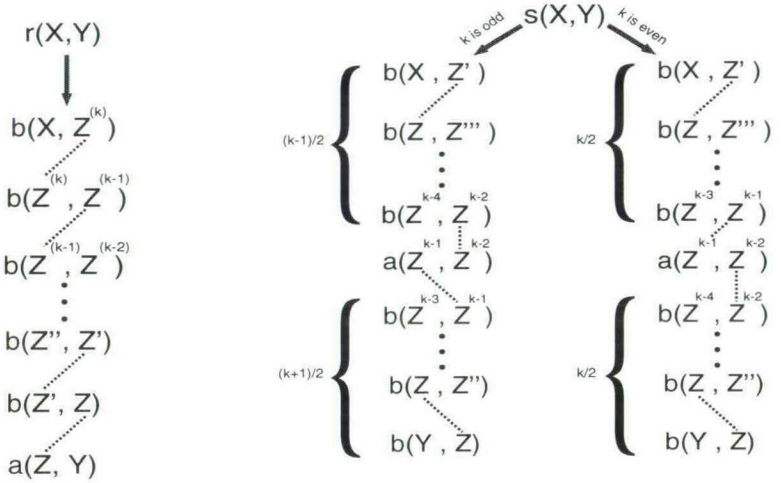


Figure 5.5: Two cases of linear recursion

gives an idea what happens when an insertion in a or b appears. In case of an insertion U in a with respect to relation r U starts new chains. Note that an update expression must be an expression that describes these new chains. Obviously a new chain is U itself. Also other chains may be present depending on the transitive closure of b.

DEFINITION 5.19 Let $r(X, Y) \longleftarrow p(X, Z), r(Z, Y)$ be the recursive part of the definition defining a binary recursive predicate r, given some binary relation p. The *transitive closure* of p is represented by r_p and is defined by:

$$r_p(X, Y) \longleftarrow p(X, Y)$$

$$r_p(X, Y) \longleftarrow p(X, Z), r_p(Z, Y)$$

The formulation of update expressions by using transitive closure operations is elaborated in the next section.

In [JAN87] it is shown under which conditions and how a general linear recursive rule can be expressed by a transitive closure operation. Transitive closure operations are efficiently computable (see [DR94]) and therefore are important for using linear recursive rules in practice.

5.3.3 Linear Recursion in Inconsistency Rules

Introducing linear recursion in the method based on inconsistency rules is not so straightforward as the particular linear recursion of EXAMPLE 5.11 makes us believe. The previous section indicated that the construction of update expressions may be rather difficult when the variables in the recursive definition do not contain a certain regularity in the order of variables as in the case of the *ancestor* predicate. Further, more than one base relation may be involved in defining the recursive relation, which makes update expressions more complex.

5.3.3.1 Linear Recursion in Update Expressions

The update expression with respect to a recursive predicate uses the transitive closure of the predicate in its definition. This is illustrated by the following example.

EXAMPLE 5.12 Consider the definition of r by the rules R_1 and R_2 in the previous section. First, we are interested in the update expression of r with respect to insertions in a, i. e., $\Delta^{r(X,Y)}_{a(Z,Y)}$. The new chains after an insertion in a will depend on r_b and are described by the following expressions:

$$ins(r(X, Y)) \longleftarrow r_b(X, Z), ins(a(Z, Y))$$

$$ins(r(X, Y)) \longleftarrow ins(a(X, Y))$$

or more concisely expressed by

$$ins(r(X, Y)) \longleftarrow (Z = X\ ;\ r_b(X, Z)), ins(a(Z, Y)).$$

So, the update expression $\Delta^{r(X,Y)}_{a(Z,Y)}$ is equal to:

$$new((Z = X\ ;\ r_b(X, Z)))$$

Next, we are interested in the update expression of r with respect to insertions in b, i. e., $\Delta^{r(X,Y)}_{b(V,W)}$. Note that when b is updated, r_b is updated like the *ancestor* predicate by an update in *parent*. It is easy to see that

$$\Delta^{r(X,Y)}_{b(V,W)} = \Delta^{r_b(X,Z)}_{b(V,W)}, \Delta^{r(X,Y)}_{r_b(X,Z)} = \Delta^{r_b(X,Z)}_{b(V,W)}, new(a(Z, Y)).$$

Note that it is already known from EXAMPLE 5.11 that $\Delta^{r_b(X,Z)}_{b(V,W)}$ is equal to:

$$new((r_b(X, V)\ ;\ X = V), (r_b(W, Z)\ ;\ Z = W))$$

So, the update expression $\Delta^{r(X,Y)}_{b(V,W)}$ is equal to:

$$new((r_b(X, V)\ ;\ X = V), (r_b(W, Z)\ ;\ Z = W)), new(a(Z, Y))$$

REMARK The update expressions for recursion should be expressed by using transitive closure expressions of the base relations used in the recursive definition. Therefore, the transformation of general recursive rules into linear recursive rules expressed by transitive closure is encouraged from the integrity checking point of view.

REMARK In the case of s updates in a and b can create more complex update expressions than in the case of r, which cannot be expressed by transitive closure expressions r_b since the number of b-facts in a chain before and after the appearence of a fact a are related.

5.3.3.2 Ordered Linear Recursive Predicates in Update Expressions

By ordered linear recursive predicates we mean those recursive definitions that can be pictured by a single chain of literals like in the case of $r(X, Y)$ of FIGURE 5.5. However, in general the chain can be more hidden than in this particular case, because the number of variables and the number of relations in defining the recursive predicate can vary. In this section, for these general kinds of ordered linear recursive predicates update expressions are derived.

Consider the following general linear recursive definition of r:

R_1: $r(\bar{X}, \bar{Y}) \longleftarrow \bar{a}(\bar{X}, \bar{Y})$

R_2: $r(\bar{X}, \bar{Y}) \longleftarrow \bar{b}(\bar{X}, \bar{Z}), r(\bar{Z}, \bar{Y})$

where $\bar{X}$, $\bar{Y}$ and $\bar{Z}$ are sequences of variables and $\bar{a}$ and $\bar{b}$ represent some conjunctions of literals each containing some, not necessary all, of the variables that appear in their arguments. However, we suppose the rules are range-restricted, which means that all variables in $\bar{X}$ and $\bar{Y}$ must appear somewhere in the body of the rule. Note that in R_2 the distinguished variables in the head of the definition are introduced in the same order as in the body of this rule. This is a more restricted form compared to the restrictions on the variables in the definition of r in 5.3.2. Further note that in R_2 the number of variables in $\bar{X}$ must be equal to the number in $\bar{Z}$. It turns out that for these so called *ordered linear recursive predicates* update expressions can easily be formulated, comparable to those derived in EXAMPLE 5.12 for a simple case of a linear recursive predicate with ordered variables. However, we need a more general notion of Δ. Instead of $\Delta^{r(\bar{X},\bar{Y})}_{a(\bar{Z},\bar{Y})}$, where a is a single predicate, an expression $\Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}$ is introduced where $\bar{a}$ is a conjunction of literals. The idea of Δ^L_C is still applicable. This expression states the evaluation that has to be performed to find the updates in L caused by an update in C. More precisely, $\Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}$ is the expression that determines the insertions in $r(\bar{X}, \bar{Y})$ when an insertion in $\bar{a}(\bar{Z}, \bar{Y})$ appears. Note that an insertion in $\bar{a}(\bar{Z}, \bar{Y})$ depends on the updates of each of the conjuncts of $\bar{a}$. The same considerations apply to the expression $\Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}$.

Now, consider a recursively defined relation r that is defined by using a conjunction of literals containing relation c. Until now we supposed that the relation c was a base relation, but what happens when this relation is derived? Suppose one of the base relations that were used to define c is the m-ary relation c_0. An update to c_0 could lead to updates of the relation c. In turn, the induced

updates in c have the same influence on the recursive predicate as if they were directly updated. Let c be a predicate having the sequence $\bar{U}$ as its arguments. This sequence of variables may contain variables that also appear in the remainder of the rule in which c occurs. The only difference is that instead of $ins(c(\bar{U}))$ we have an expression $ins(c_0(C_1, C_2, \ldots, C_m))$, $\Delta^{c(\bar{U})}_{c_0(C_1,C_2,\ldots,C_m)}$. Suppose $\Delta^{r(\bar{X},\bar{Y})}_{c(U_1,U_2,\ldots,U_n)}$ can be easily derived, then the update expression $\Delta^{r(\bar{X},\bar{Y})}_{c_0(C_1,C_2,\ldots,C_m)}$ can be expressed by the update expressions of r and c and of c and c_0. So, the update expression of a base relation leading to a derived predicate that is used in a definition of a linear recursive predicate can be derived by the composition of the two update expressions. Suppose $r(\bar{X}, \bar{Y})$ appears in an inconsistency indicator II, then the following inconsistency rule for an update in c_0 leading to II is derived:

$$inconsistent(ins(c_0(C_1, C_2, \ldots, C_m)) \Leftarrow \Delta^{c(\bar{U})}_{c_0(C_1,C_2,\ldots,C_m)}, \Delta^{r(\bar{X},\bar{Y})}_{c(\bar{U})}, new(II_{r(\bar{X},\bar{Y})})$$

Suppose $c(\bar{U})$ appears in $\bar{a}$ or $\bar{b}$, which are used in defining the linear recursive predicate r like before. $\Delta^{r(\bar{X},\bar{Y})}_{c(\bar{U})}$ is expressed by $\Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}$, when $c(\bar{U})$ appears in $\bar{a}$, or $\Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}$, when $c(\bar{U})$ appears in $\bar{b}$. In turn, both expressions are expressible by transitively closing $\bar{b}$ which is defined in the following definition which is a generalization of DEFINITION 5.19.

DEFINITION 5.20 Let $r(\bar{X}, \bar{Y}) \longleftarrow \bar{p}(\bar{X}, \bar{Z}), r(\bar{Z}, \bar{Y})$ be the recursive part of the definition defining a recursive predicate r, for some relations $\bar{p}$. The *transitive closure* of $\bar{p}$ is represented by $r_{\bar{p}}$, and defined by:

$$r_{\bar{p}}(\bar{X}, \bar{Y}) \longleftarrow \bar{p}(\bar{X}, \bar{Y})$$
$$r_{\bar{p}}(\bar{X}, \bar{Y}) \longleftarrow \bar{p}(\bar{X}, \bar{Z}), r_{\bar{p}}(\bar{Z}, \bar{Y})$$

In the expressions above, $\Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}$ and $\Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}$ are generalizations of the expressions derived in EXAMPLE 5.12. Similarly, we find that $\Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}$ is equal to:

$$new((\bar{Z} = \bar{X}\ ;\ r_{\bar{b}}(\bar{X}, \bar{Z}))),$$

where $\bar{Z} = \bar{X}$ is interpreted as $Z_j = X_j$ for each Z_j in the sequence $\bar{Z}$ and each X_j in the sequence $\bar{X}$, for each j.

Also we find that $\Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})} = \Delta^{r_{\bar{b}}(\bar{X},\bar{Z})}_{\bar{b}(\bar{V},\bar{W})}, \Delta^{r(\bar{X},\bar{Y})}_{r_{\bar{b}}(\bar{X},\bar{Z})} =$

$$\Delta^{r_{\bar{b}}(\bar{X},\bar{Z})}_{\bar{b}(\bar{V},\bar{W})}, new(\bar{a}(\bar{Z}, \bar{Y})) =$$

$$new((r_{\bar{b}}(\bar{X}, \bar{V})\ ;\ \bar{X} = \bar{V}), (r_{\bar{b}}(\bar{W}, \bar{Z})\ ;\ \bar{Z} = \bar{W})), new(\bar{a}(\bar{Z}, \bar{Y})).$$

Suppose c appears in $\bar{a}$. Then by interpreting $\Delta^{\bar{a}(\bar{Z},\bar{Y})}_{c(\bar{U})}$ as before, an insertion in $c(\bar{U})$ implies an insertion in $\bar{a}(\bar{Z}, \bar{Y})$ if the conjunction of literals in $\bar{a}(\bar{Z}, \bar{Y})$ where $c(\bar{U})$ is left out, expressed by $\bar{a}(\bar{Z}, \bar{Y})_{c(\bar{U})}$, holds in the updated database. So,

$$\Delta^{r(\bar{X},\bar{Y})}_{c(\bar{U})} = \Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}, \Delta^{\bar{a}(\bar{Z},\bar{Y})}_{c(\bar{U})} = \Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}, new(\bar{a}(\bar{Z}, \bar{Y})_{c(\bar{U})}).$$

In this case the revised inconsistency rule for an insertion in c_0 is:

$$inconsistent(ins(c_0(C_1, C_2, \ldots, C_m)) \Leftarrow$$
$$\Delta^{c(\bar{U})}_{c_0(C_1,C_2,\ldots,C_m)}, \Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}, new(\bar{a}(\bar{Z}, \bar{Y})_{c(\bar{U})}), new(II_{r(\bar{X},\bar{Y})})$$

Using the above expression for $\Delta^{r(\bar{X},\bar{Y})}_{\bar{a}(\bar{Z},\bar{Y})}$, this revised inconsistency rule can be expressed by:

$$inconsistent(ins(c_0(C_1, C_2, \ldots, C_m)) \Leftarrow$$
$$\Delta^{c(\bar{U})}_{c_0(C_1,C_2,\ldots,C_m)}, new((\bar{Z} = \bar{X}\ ;\ r_{\bar{b}}(\bar{X}, \bar{Z}))), new(\bar{a}(\bar{Z}, \bar{Y})_{c(\bar{U})}), new(II_{r(\bar{X},\bar{Y})})$$

Similarly, when c appeared in $\bar{b}(\bar{X}, \bar{Z})$ we would have found that

$$\Delta^{r(\bar{X},\bar{Y})}_{c(\bar{U})} = \Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}, \Delta^{\bar{b}(\bar{V},\bar{W})}_{c(\bar{U})} = \Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}, new(\bar{b}(\bar{V}, \bar{W})_{c(\bar{U})}).$$

In this case the revised inconsistency rule is:

$$inconsistent(ins(c_0(C_1, C_2, \ldots, C_m)) \Leftarrow$$
$$\Delta^{c(\bar{U})}_{c_0(C_1,C_2,\ldots,C_m)}, \Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}, new(\bar{b}(\bar{V}, \bar{W})_{c(\bar{U})}), new(II_{r(\bar{X},\bar{Y})})$$

Using the above expression for $\Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}$, this revised inconsistency rule can be expressed by:

$$inconsistent(ins(c_0(C_1, C_2, \ldots, C_m)) \Leftarrow$$
$$\Delta^{c(\bar{U})}_{c_0(C_1,C_2,\ldots,C_m)}, new((r_{\bar{b}}(\bar{X}, \bar{V})\ ;\ \bar{X} = \bar{V}), (r_{\bar{b}}(\bar{W}, \bar{Z})\ ;\ \bar{Z} = \bar{W})),$$
$$new(\bar{a}(\bar{Z}, \bar{Y})), new(\bar{b}(\bar{V}, \bar{W})_{c(\bar{U})}), new(II_{r(\bar{X},\bar{Y})})$$

REMARK Suppose that $\bar{a} = \bar{b}$. Then we get a special case which, for instance, is the case in the *ancestor* definition. In this case, the recursive predicate r is equal to the transitive closure $r_{\bar{b}}$. Note that in this case $\Delta^{r(\bar{X},\bar{Y})}_{\bar{b}(\bar{V},\bar{W})}$ reduces to

$$new((r(\bar{X}, \bar{V})\ ;\ \bar{X} = \bar{V}), (r(\bar{W}, \bar{Y})\ ;\ \bar{Y} = \bar{W})).$$

In this case the revised inconsistency rule reduces to:

$$inconsistent(ins(c_0(C_1, C_2, \ldots, C_m)) \Leftarrow$$
$$\Delta^{c(\bar{U})}_{c_0(C_1,C_2,\ldots,C_m)}, new((r(\bar{X}, \bar{V})\ ;\ \bar{X} = \bar{V}), (r(\bar{W}, \bar{Y})\ ;\ \bar{Y} = \bar{W})),$$
$$new(\bar{b}(\bar{V}, \bar{W})_{c(\bar{U})}), new(II_{r(\bar{X},\bar{Y})})$$

EXAMPLE 5.13 Let D be a database consisting of the following rules and inconsistency indicator:

R_1: $investor(X, C) \longleftarrow shareholder(X, N, C), N \geq 10$

R_2: $family\_company(X, Y, C) \longleftarrow parent(X, Y), investor(X, C), investor(Y, C)$

R_3: $family\_company(X, Y, C) \longleftarrow parent(X, Z), investor(X, C), family\_company(Z, Y, C)$

R_4: $parent(X, Y) \longleftarrow father(X, Y)$

R_5: $parent(X, Y) \longleftarrow mother(X, Y)$

R_6: $mother(X, Y) \longleftarrow husband(Z, X), father(Z, Y)$

II : $family\_company(X, Y, C), government(C)$

These rules reflect the world's observation that two people are owners of a family company if these two people are both investors of the company, i. e. , shareholders possessing at least ten percent of the company's shares, and are blood related. A secondary condition is that in each generation of these family members there must be a member who is also an investor of the company. The inconsistency indicator states that an inconsistency occurs when the a company is owned by the government and at the same time is a family company. Several base relations are involved. The base relation $shareholder(X, N, C)$ expresses that a person X has N percent of the shares of company C. The base relation $government(C)$ expresses that a company C is owned by the government. Also we have the well known base relations *husband* and *father*. Suppose we want to obtain the revised inconsistency rules for an insertion in *husband*. In order to generate the revised inconsistency rules we have to derive $\Delta^{parent(Z,Y)}_{husband(W,Z)}$, $\Delta^{parent(V,W)}_{husband(Z',V)}$ and $II_{family\_company(X,Y,C)}$, which are equal to $new(father(W, Y))$, $new(father(Z', W))$ and $new(government(C))$ respectively.

Let us first concentrate on the derivation of the revised inconsistency rule that is built by the use of $parent(X, Y)$ appearing in R_2. For generating this inconsistency rule the update expression $\Delta^{family\_company(X,Y,C)}_{parent(Z,Y)}$ for $parent(X, Y)$ appearing in R_2 is important. So, this update expression is decomposed in

$$\Delta^{family\_company(X,Y,C)}_{parent(Z,Y),investor(Z,C),investor(Y,C)},$$

which is equal to $new((Z = X\ ;\ tc(X, Z, C)))$, and

$$\Delta^{parent(Z,Y),investor(Z,C),investor(Y,C)}_{parent(Z,Y)},$$

which is equal to $new(investor(Z, C), investor(Y, C))$.

Here, $tc(X,Z,C)$ is the transitive closure of $parent(X, Z)$, $investor(X, C)$ in the recursive definition of *family_company*, namely:

$tc(X, Y, C)) \longleftarrow parent(X, Y), investor(X, C)$

$tc(X, Y, C)) \longleftarrow parent(X, Z), investor(X, C), tc(Z, Y, C))$

Now, by using the remarks above the following equality is derived:

$$\Delta^{family\_company(X,Y,C)}_{parent(Z,Y)} = new((Z = X\ ;\ tc(X, Z, C))), new(investor(Z, C), investor(Y, C)).$$

Combined with the update expression $\Delta^{parent(Z,Y)}_{husband(W,Z)}$ and $II_{family\_company(X,Y,C)}$ the following revised inconsistency rule is derived:

$$inconsistent(ins(husband(W, Z))) \Leftarrow$$
$$new(father(W, Y)), new((Z = X\ ;\ tc(X, Z, C)),$$
$$new(investor(Z, C), investor(Y, C)), new(government(C))$$

When concentrating on the derivation of the revised inconsistency rule that is built by the use of $parent(X, Y)$ appearing in R_3, the update expression $\Delta^{family\_company(X,Y,C)}_{parent(V,W)}$ for $parent$(X,Z) appearing in R_3 is important. This update expression is decomposed in

$$\Delta^{family\_company(X,Y,C)}_{parent(V,W),investor(V,C),investor(W,C)},$$

which is equal to

$$new((tc(X, V, C) \; ; \; X = V), (tc(W, Z, C) \; ; \; Z = W)), new(parent(Z, Y)),$$
$$new(investor(Z, C), investor(Y, C)))$$

and

$$\Delta^{parent(V,W),investor(V,C),investor(W,C)}_{parent(V,W)},$$

which is equal to $new(investor(V, C), investor(W, C))$.

So, $\Delta^{family\_company(X,Y,C)}_{parent(V,W)}$ is equal to
$new((tc(X, V, C) \; ; \; X = V), (tc(W, Z, C) \; ; \; Z = W)),$
$new(investor(V, C), investor(W, C)).$

Combined with the update expression $\Delta^{parent(V,W)}_{husband(Z',V)}$ and $II_{family\_company(X,Y,C)}$ the following revised inconsistency rule is derived:

$$inconsistent(ins(husband(Z', V))) \Leftarrow$$
$$new(father(Z', W)), new((tc(X, V, C) \; ; \; X = V), (tc(W, Z, C) \; ; \; Z = W)),$$
$$new(investor(V, C), investor(W, C)), new(government(C))$$

5.4 Complexity in $\mathcal{FICCS}$

As we have noted in the beginning of this thesis the range-restricted , allowed or some other property, which guarantees domain independent query answering, is important to be able to compute the answer of a query in a feasible time in the number of facts of the database. A more precise result in [Wüe91] states that this computation of an answer of a query in a deductive database can be done in polynomial time when the query and the database are constrained.

PROPOSITION 10 *Let D be a finite deductive database such that each rule, each constraint and each query Q to D is allowed. Let n be the number of facts and l be the maximum length, e. g., the number of characters, of the set of formulas in D and a query. Then it holds that*

(i) the answer to Q can be computed in time $O(n^{3*l})$.

(ii) the consistency of the deductive database can be decided in time $O(n^{3*l})$.

Because in the proposed method the consistency check is presented as a query the consistency check is also decidable in polynomial time in the number of facts of the database, if the length of the revised inconsistency rules is constrained as well.

References

[BDM87] FRANÇOIS BRY, HENDRIK DECKER AND RAINER MANTHEY. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In J. W. SCHMIDT, S. CERI AND M. MISSIKOFF, editors, *Advances in Databases Technology, EDBT '88; Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 488–505, Venice, Italy, November 1987. also: ECRC Technical Report KB-16.

[Deß93] STEFAN DESSLOCH. *Semantic Integrity in Advanced Database Management Systems*. PhD thesis, Dissertation at the University of Informatics of Kaiserslautern, 1993.

[DKM91] C. DELOBEL, M. KIFER AND Y. MASUNAGA, editors. *Deductive and Object-Oriented Databases; Proceedings of the Second International Conference, DOOD'91*, volume 566 of *Lecture Notes in Computer Science*, Munich, Germany, December 1991.

[Don92] G. DONG. On Datalog Linearization of Chain Queries. *Theoretical Studies in Computer Science*, pages 181–206, 1992.

[DR94] SHAUL DAR AND RAGHU RAMAKRISHNAN. A Performance Study of Transitive Closure Algorithms. In RICHARD T. SNODGRASS AND MARIANNE WINSLETT, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23-2, pages 454–465, Minneapolis, Minnesota, May 1994. ACM Press.

[DT87] UMESHWAR DAYAL AND IRV TRAIGER, editors. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 16-3, San Francisco, May 1987. ACM Press.

[GP95] IRÈNE GUESSARIAN AND JEAN-ERIC PIN. Linearizing Some Recursive Logic Programs. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):137–149, February 1995.

[HZL93] JIAWEI HAN, KANGSHENG ZENG AND TONG LU. Normalization of Linear Recursions in Deductive Databases. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 559–567, Vienna, Austria, April 1993.

[IW89] YANNIS E. IOANNIDIS AND EUGENE WONG. Transforming Nonlinear Recursion into Linear Recursion. pages 401–422, Vienna, Virginia, USA, April 1989. Benjamin Cummings.

[JAN87] H. V. JAGADISH, RAKESH AGRAWAL AND LINDA NESS. A Study of Transitive Closure as a Recursive Mechanism. In Dayal and Traiger [DT87], pages 331–344.

[Jeu92] MANFRED JEUSFELD. *Änderungskontrolle in Deduktiven Objektbanken*. PhD thesis, University of Passau, Germany, 1992.

[JJ91] MANFRED JEUSFELD AND MATTHIAS JARKE. From Relational to Object-Oriented Integrity Simplification. In Delobel et al. [DKM91], pages 460–477.

[JK90] MANFRED JEUSFELD AND EVA KRÜGER. Deductive Integrity Maintenance in an Object-Oriented Setting. MIP 9013, Technische Berichte der Fakultät für Mathematik und Informatik Universität Passau, 1990.

[Kog95] E. A. DE KOGEL. *Equational proofs in Tableaux and Logic Programming*. PhD thesis, Tilburg, The Netherlands, 1995.

[LLH94] WENYU LU, DIK LUN LEE AND JIAWEI HAN. A Study on the Structure of Linear Recursion. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):723–737, 1994.

[Nau86] JEFFREY F. NAUGHTON. Data Dependent Recursion in Deductive Databases. In *Proceedings of the Fifth ACM SIGART-SIGMOD Symposium on Principles of Database Systems*, volume II, pages 267–279, March 1986.

[Nau89] JEFFREY F. NAUGHTON. Data Dependent Recursion in Deductive Databases. *Journal of Computer and System Sciences*, 38:259–289, 1989.

[Oph92] W. M. J. OPHELDERS. *Automated Theorem Proving based upon a Tableau-method with Unification under Restrictions: Theory, Implementation and Empirical Results*. PhD thesis, Department of Philosophy, KUB Tilbury, Tilburg, The Netherlands, 1992.

[Sar89] Y. SARAIYA. Linearizing Nonlinear Recursions in Polynomial Time. In *ACM Symposium on Principles of Database Systems (PODS)*, volume 13 of *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 182–189. ACM, June 1989.

[SO93] H. C. M. DE SWART AND W. M. J. OPHELDERS. Tableaux versus Resolution; a Comparison. *Fundamentae Informaticae*, 18:109–127.

[STW93] KLAUS-DIETER SCHEWE, BERNHARD THALHEIM AND INGRID WETZEL. Integrity Preserving Updates in Object-Oriented Databases. In M. E. ORLOWSKA AND M. PAPAZOGLOU, editors, *Advances in Database Research; Proceedings of the Fourth Australian Database Conference*, pages 171–185, Brisbane, Australia, February 1993.

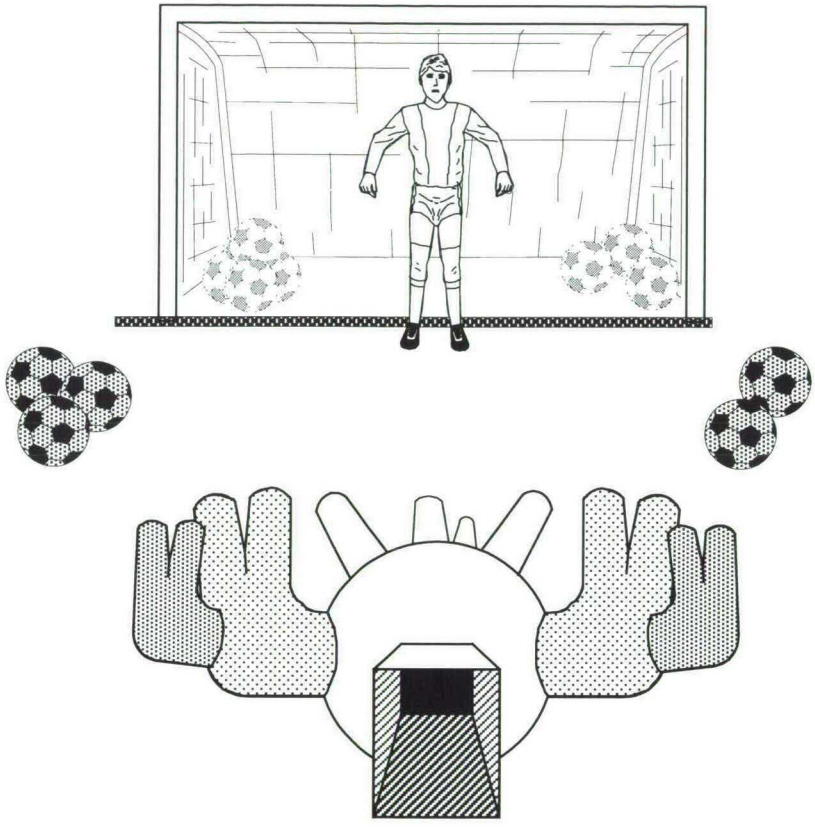
[Ull91] J. D. ULLMAN. A Comparison of Deductive and Object-Oriented Database Systems. In Delobel et al. [DKM91], pages 263–277.

[Wüe91] BEAT WÜETHRICH. *Large Deductive Databases with Constraints.* PhD thesis, Swiss Federal Institute of Technology Zürich, 1991.

[YKHH92] CHEONG YOUN, HYOUNG-JOO KIM, LAWRENCE J. HENSCHEN AND JIAWEI HAN. Classification and Compilation of Linear Recursive Queries in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(1):52–67, October 1992.

[ZY87] WEINING ZHANG AND C. T. YU. A Necessary Condition for a Doubly Recursive Rule to be Equivalent to a Linear Recursive Rule. In Dayal and Traiger [DT87], pages 345–356.

"... but the fact hoped that all of its induced facts would pass the integrity check."



Chapter 6

Design and Implementation of $\mathcal{FICCS}$

In this chapter, the integration of $\mathcal{FICCS}$ and a deductive database system is described. This chapter is divided into two sections. In the first section, the design issues of $\mathcal{FICCS}$ are given. It presents a functional decomposition of $\mathcal{FICCS}$. Further, the software components that were chosen to implement $\mathcal{FICCS}$ are presented. In the second section, an implementation of $\mathcal{FICCS}$ is given. In this section, the implementation is constrained to only the non-system oriented part of $\mathcal{FICCS}$. So, system calls for the creation of input/output forms and calls to the transaction manager of the underlying deductive database management system after an integrity check are not elaborated here.

6.1 Design of $\mathcal{FICCS}$

In $\mathcal{FICCS}$ three main components can be distinguished. Each component has one particular task. These components model the first level of the architecture of $\mathcal{FICCS}$. The first component is concerned with the control of $\mathcal{FICCS}$. The second component is responsible for loading a deductive database. The Input/Output facilities in this component depend on the system and are not interesting when looking at the integrity checking task of the system. Therefore, they are not elaborated in detail. The third component is the core of $\mathcal{FICCS}$ and will be decomposed until the implementational level is reached.

6.1.1 An Overview of Components of $\mathcal{FICCS}$

In this section, both the interaction between the three main components and the internal structure of the first and second component are described. The interaction between the three main components, the user and the deductive database management system is schematically represented by FIGURE 6.1. First, we deal with a component that is responsible for the control of $\mathcal{FICCS}$. This component loads the proper files containing $\mathcal{FICCS}$ and loads additional files containing code that is used by $\mathcal{FICCS}$, such as predicates that are responsible for the input and output of the data of the program, general auxiliary predicates and predicates that were responsible for establishing a proper coupling to a database management system. Further, it is responsible for switching the

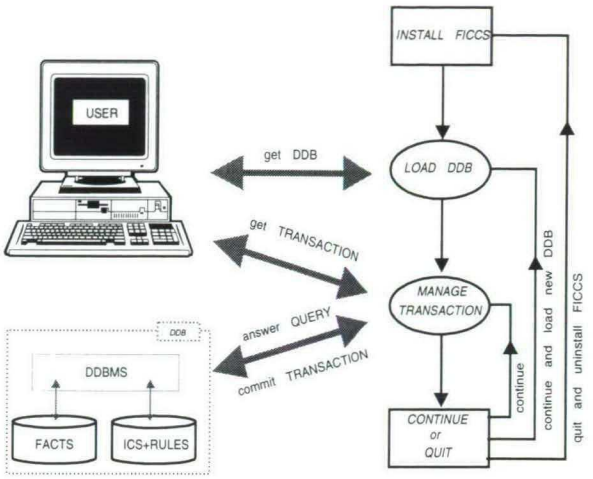


Figure 6.1: The main decomposition of *FICCS*

control to one of the other two components. After each integrity check, three cases can occur, namely

- the user wants to continue using another transaction with the current deductive database,
- the user wants to continue with another deductive database,
- the user wants to quit *FICCS*.

The second component is responsible for loading a proper deductive database state and for the proper attachment of *FICCS* with the deductive database management system. This component responsible for loading a deductive database is divided in three subcomponents, i. e. , one responsible for loading the extensional database, one responsible for loading the intensional database and one for obtaining the revised inconsistency rules. In the first two subcomponents, the user will be asked to select the files containing the extensional and intensional database, after which they will be loaded. When the sets of rules and inconsistency indicators are obtained, the third subcompo nent is responsible for delivering the proper set of related revised inconsistency rules. When the user does not deliver a file containing the proper revised inconsistency rules, these revised inconsistency rules are generated from scratch from the set of rules and inconsistency indicators, calling the revised inconsistency rule generator. The implementation of this revised inconsistency rule generator can be found in 6.2.4. These subcomponents are illustrated in FIGURE 6.2.

The third component that is responsible for the management of the transaction and which is the most interesting one from the constraint checking point of view is elaborated in 6.1.2.

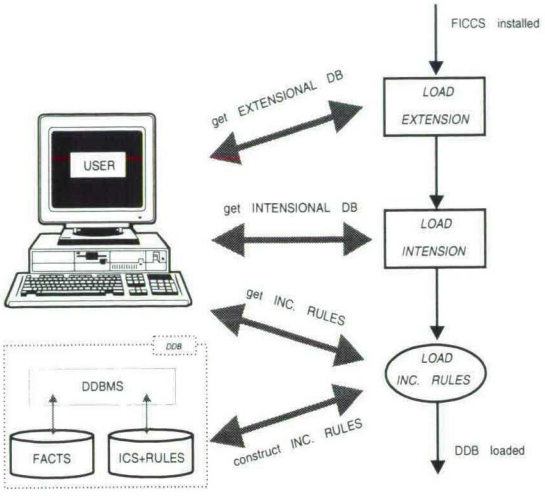


Figure 6.2: The decomposition of the component that loads the DDB.

6.1.2 Transaction Management in $\mathcal{FICCS}$

The third component of $\mathcal{FICCS}$, which is responsible for the management of the transaction, is divided in three subcomponents. The interaction between these subcomponents, the user and the deductive database management system is represented in FIGURE 6.3.

The first subcomponent is responsible for obtaining the transaction from the user and converting it into the proper format before applying it to the set of revised inconsistency rules. The second subcomponent is responsible for adjusting the set of revised inconsistency rules, when the set of rules and/or integrity constraints is changed. The third subcomponent is responsible for the application of the revised inconsistency rules and the transfer of the necessary queries to the query evaluator. In turn, the query evaluator must evaluate those queries properly, i.e., in either the current or the updated database. When the query evaluator has given back the control to this subcomponent by returning an answer whether the queries have led to an inconsistency or not, the subcomponent based on this answer takes the proper actions. If the integrity check does not detect some inconsistency, then the subcomponent asks the user if it has to commit or to rollback the current transaction, else it rollbacks the transaction without any interaction with the user. The implementation of the component responsible for managing transactions is given in 6.2.2.3.

6.1.3 Design Model of $\mathcal{FICCS}$

In this section, the design model of the system $\mathcal{FICCS}$ is given. The whole method including the concept of revised inconsistency rules can easily be implemented by using the logical application

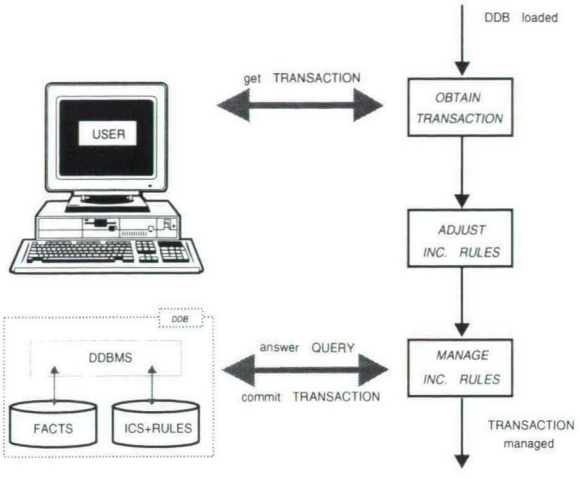


Figure 6.3: The decomposition of the component that manages a transaction

language used by a deductive database management system. The databases used in the examples in this thesis can be loaded easily in the main memory of the logical application language. In that case, the integrity check runs in the main memory of the logical application language only. However, when the number of facts exceeds the memory of the logical application language, the data must be on secondary storage. In case of a fact base on secondary storage, rules and inconsistency indicators are loaded into the memory of the deductive database management system, while the facts are accessible by the proper attachment to the fact base of the deductive database management system. However, the system is responsible for data as well as rule and inconsistency indicator management. This also implies that the system can handle queries containing derived relations. When $\mathcal{FICCS}$ is implemented in the logical application language of the deductive database management system, it does not need a special query evaluator, but it can use the query evaluator of the deductive database management system. In most cases, the logical application language is a Prolog-like language.

In this thesis, another approach is chosen to implement $\mathcal{FICCS}$. Although the implementation of $\mathcal{FICCS}$ in the logical application language of the used deductive database management system is preferable, we have chosen to implement $\mathcal{FICCS}$ in Prolog, coupled to a relational database management system. This choice is influenced by the fact that most people are familiar with relational systems, while they are not familiar with deductive systems. This choice implies that the interaction of $\mathcal{FICCS}$ with its environment, as depicted by the previous figures in this chapter, is changed by replacing the interaction with a deductive system by the interaction with a relational system. This is illustrated in FIGURE 6.4. By giving the implementation for the coupling with a relational system, the benefit of deductive systems in favour of relational systems extended

with rules may once again be clear, because of the greater transparency. For, the management of rules and inconsistency indicators is a responsibility of the deductive database management system and queries containing derived relations can be handled by the query evaluator of the deductive database management system. When using a coupling of Prolog to a relational database

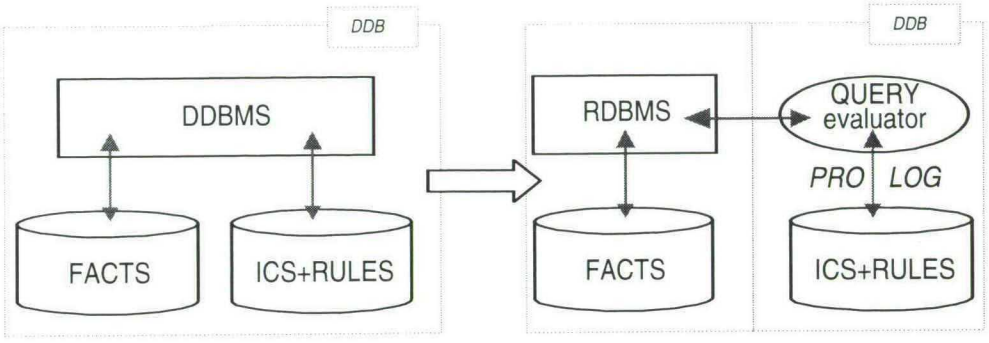


Figure 6.4: A deductive database system built from relational database system

management system, only the query evaluator of the relational database management system can be used. Hence, the query evaluator can only obtain data from tables efficiently. However, the relational database management system is insufficient for handling rules and inconsistency indicators; so, the management and evaluation of rules and inconsistency indicators have to be handled by Prolog.

A tight coupling of Prolog to a relational database system is established by using special purpose software for the communication between Prolog and the relational database management system. The whole system, i. e. , a deductive database system with integrity constraint checking capabilities, is built from three software components:

- LPA-Prolog for windows,
- a Database Interface (DBI)-toolkit by LPA,
- Q & E database library from Pioneer Software, Inc..

The Logic Programming Associates (LPA) provides the logical programming language Prolog, which runs under windows. This Prolog offers the possibility to access a large set of Graphical User Interface (GUI) functions of Windows, allowing the user to create full window applications. LPA provide also a Database Interface (DBI)-tool, which is able to interface the Q & E database library. This database library, manufactured by Pioneer Software, Inc., is an interface which allows us to use datafiles of all kinds of database formats. So, the interface between the programming language and the database is built from two components, i. e. , the DBI and the Q & E database library. The communication language in the Q & E database library is SQL. The DBI is only responsible for the translation of a Prolog goal to an equivalent SQL-query. FIGURE 6.5 shows the

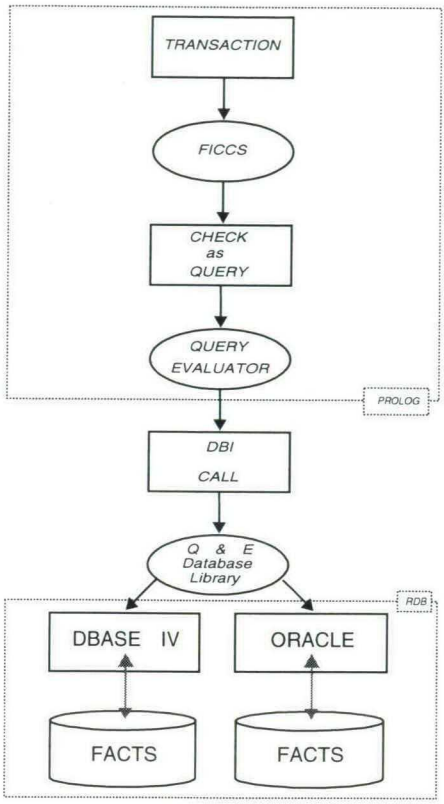


Figure 6.5: A deductive database system with integrity constraint checking.

global architecture of the system. *FICCS* is implemented by using these software components. The database rules and constraints are implemented in LPA-Prolog, while the facts are stored by a relational database management system, for instance DBASE or ORACLE. Input/Output forms are implemented by using the GUI of Windows, while the database query handling is done efficiently by the DBI-tool and the Q & E database library. How the Input/Output forms are implemented is closely related to the taste of the user and the programmer of the final system and therefore will not be elaborated here. Only the implementation of the core of *FICCS* is given, i. e. , the implementation of the method based on revised inconsistency rules.

6.2 Implementation of $\mathcal{FICCS}$

In the previous section, a functional decomposition of $\mathcal{FICCS}$ was given. In this section, the implementation of several of the most interesting parts of $\mathcal{FICCS}$ is given. In order to get a better comprehension of the implementation, the most frequently used data structures and an often used auxiliary predicate running on lists are elaborated. Further, this section presents an implementation of the query evaluator, which was used in this specific implementation of $\mathcal{FICCS}$ and it presents a revised inconsistency rule generator, which is used by $\mathcal{FICCS}$ to automatically generate the revised inconsistency rules for any deductive database.

6.2.1 Primitive Concepts in $\mathcal{FICCS}$

The implementation of $\mathcal{FICCS}$ contains several primitive concepts. For instance, some variables in the implementation represent primitive data structures. Further, an auxiliary predicate for determining properties of members of lists is defined, which frequently occurs. This predicate is used as if it was a primitive predicate. The formats of the most important data structures as well as the idea behind the most frequently used auxiliary predicate are given.

6.2.1.1 Data Structures

A format of a data structure is a meta-expression representing some expression in the object language of Prolog and is recognized by putting the expression between the two parenthesis ⟨ and ⟩. When the symbols are in `typewriter` font, these symbols are part of the object language of the implementation. Further, formats may contain the meta-symbols | and $*$. Here, | represents the possibility of choice, the format on the left hand side of $\equiv$ is equal to either the format on the left side or on the right side of |. The asterisk $*$ represents an arbitrary sequence of the format with length varying from zero to any natural number. All other characters are part of the object language of the implementation. For instance, \+ is part of the object language and represents negation in Prolog. The formats of the primitive structures are:

⟨*fact*⟩ represents a fact

⟨*atom*⟩ represents an atom

⟨*literal*⟩ $\equiv$ \+⟨*atom*⟩ | ⟨*atom*⟩

⟨*conjunction*⟩ $\equiv$ [⟨*literals*⟩*]

⟨*var*⟩ $\equiv$ *var* is a variable

⟨*ID*⟩ $\equiv$ *ID* is a natural number.

Note that in the implementation a conjunction is described by a list of literals enclosed by brackets, while in the theory the literals where enclosed by parentheses. By using these primitive structures, some frequently used compound structures in the implementation are defined.

A rule, denoted for short as ⟨*rule*⟩, is represented by:

`rule(`⟨*rule id*⟩`, head(`⟨*atom*⟩`), body(`⟨*conjunction*⟩`)),`

where ⟨*rule id*⟩ represents the identifier of the rule `rule_id`(⟨*ID*⟩), `head`(⟨*atom*⟩) represents the head of the rule and `body`(⟨*conjunction*⟩) represents the body of the rule.

An inconsistency indicator, denoted for short as ⟨*ii*⟩, is represented by:

`ii`(⟨*ii id*⟩, ⟨*conjunction*⟩),

where ⟨*ii id*⟩ represents the inconsistency indicator identifier `ii_id`(⟨*ID*⟩) and ⟨*conjunction*⟩ represents the body of the rule.

A revised inconsistency rule, denoted for short as ⟨*incrule*⟩, is represented by:

`incrule`(⟨*incrule id*⟩, `head`(⟨*update type*⟩(⟨*update pattern*⟩)), `body`([⟨*query*⟩*])

where ⟨*incrule id*⟩ represents the revised inconsistency rule identifier `incrule_id`(⟨*ID*⟩),

⟨*update type*⟩ ≡ `ins` | `del`,

⟨*update pattern*⟩ ≡ ⟨*atom*⟩ | `rule`(`head`(⟨*atom*⟩, ⟨*var*⟩)) | `ii`(⟨*var*⟩),

and

⟨*query*⟩ ≡ `new`(⟨*literal*⟩) | `old`(⟨*literal*⟩) | `new`(⟨*conjunction*⟩) | `old`(⟨*conjunction*⟩).

Note that the body of a revised inconsistency rule contains a query.

An update, denoted for short as ⟨*update*⟩, is represented by:

⟨*update type*⟩(⟨*fact*⟩) | , ⟨*update type*⟩(⟨*rule*⟩) | ⟨*update type*⟩(⟨*ii*⟩).

The adjusted transaction reflects besides the change in the fact, rule and constraint base also the change in the set of revised inconsistency rules. It is represented by:

⟨*adjusted transaction*⟩ ≡

`adjusted`(⟨*changed facts*⟩, ⟨*changed rules*⟩, ⟨*changed iis*⟩, ⟨*changed incrules*⟩)

where

⟨*changed facts*⟩ ≡

`facts`(`new`([⟨*fact*⟩*]), `old`([⟨*fact*⟩*])),

⟨*changed rules*⟩ ≡

`rules`(`new`([⟨*rule*⟩*]), `old`([⟨*rule id*⟩*])),

⟨*changed iis*⟩ ≡

`iis`(`new`([⟨*ii*⟩*]), `old`([⟨*ii id*⟩*])),

⟨*changed incrules*⟩ ≡

`incrules(new([`⟨*incrule*⟩*`]), old([`⟨*incrule id*⟩*`])),`

while in the basic transaction ⟨*changed incrules*⟩ is not specified:

⟨*transaction*⟩ ≡

`transaction(`⟨*changed facts*⟩`,` ⟨*changed new rules*⟩`,` ⟨*changed new iis*⟩`),`

where the structures of ⟨*changed new rules*⟩ and ⟨*changed new iis*⟩ are equal to the structures of ⟨*changed rules*⟩ and ⟨*changed incrules*⟩ except that in their definitions ⟨*rule*⟩ and ⟨*ii*⟩ are replaced by ⟨*new rule*⟩ and ⟨*new ii*⟩ respectivily, which in turn are defined as:

⟨*new rule*⟩ ≡

`rule(head(`⟨*atom*⟩`), body(`⟨*conjunction*⟩`)),`

⟨*new ii*⟩ ≡

`ii(`⟨*conjunction*⟩`),`

and

⟨*new incrule*⟩ ≡

`incrule(head(`⟨*update type*⟩`(`⟨*update pattern*⟩`))), body([`⟨*query*⟩*`])).`

The structures ⟨*rule*⟩, ⟨*ii*⟩ and ⟨*incrule*⟩ are equal to respectively the structures ⟨*new rule*⟩, ⟨*new ii*⟩ and ⟨*new incrule*⟩ are equal, except that they do not possess an identifier yet. Identifiers for new rules, inconsistency indicators or revised inconsistency rules are assigned by the identifier manager, for which the implementation is given in 6.2.4.3. Note that ⟨*transaction*⟩ reflects the transaction from the user's perspective, while ⟨*adjusted transaction*⟩ reflects the transaction from the system's perspective.

Variables in the implementation can represent one of the structures represented above. In that case, the naming of the variables are in most cases strongly related to the naming of the structure they represent. For instance, the structure ⟨*transaction*⟩ is often represented by the variables called `T` or `Transaction`. Before giving the implementation of $\mathcal{FICCS}$ an additional predicate is explained, which is frequently used in the implementation.

6.2.1.2 Auxiliary Predicates in $\mathcal{FICCS}$

In the implementation an auxiliary predicate `member_list` appears. This predicate is a generalized version of the well known binary member predicate, which states that the first argument is a member of the second argument representing a list of elements. The `member_list` predicate goes beyond that. It has some additional arguments in order to be able to specify the evaluation type, the selected element of a list satisfying a certain property, which is specified as well, the position of the member in a list one wants to select, etc.. It turns out that these operations on members of lists possess a great resemblance. The full specification of the `member_list` predicate is:

`member_list(`⟨*kind_of_test*⟩`, list(`⟨*list*⟩`), mtest(`⟨*sign*⟩`, `⟨*test*⟩`), output(`⟨*list*⟩`).`

Here, ⟨*kind_of_test*⟩ can be any of the following expressions:

- `all`,
- `unique_exist`,
- `all_exist`,
- `num(`⟨*number*⟩`)`,

where ⟨*number*⟩ can represent any natural number. Further, the second argument of `member_list`, i. e., `list(`⟨*list*⟩`)`, contains as input the observed list, ⟨*sign*⟩ ≡ `pos` | `neg` and `mtest`, which is an abbreviation for membertest, tests if a member obeys a certain condition. When the first argument of `member_list` is `all`, it states that all elements of the list, which is given in the second argument, must satisfy `mtest`. When the first argument contains `all_exist`, `member_test` succeeds for each element in the list for which `mtest` is satisfied. Therefore, in this case backtracking is permitted on `member_list`. In the case of `unique_exist`, just one success of an element in the list for which `mtest` is satisfied is needed. Therefore, in this case backtracking on `member_list` after a success is avoided. When the first argument of `member_list` is `num(N)`, `member_list` succeeds only if the `N`-th element of the list satisfies `mtest`.

The last argument of `member_list` contains as output a list, which corresponds to the input list, where the currently tested member satisfying `mtest`, is deleted.

The predicate of the general format `mtest(`⟨*sign*⟩`, `⟨*test*⟩`)` will be explained further. Its first argument, i. e., ⟨*sign*⟩, can be either `pos` or `neg`. In the case of `pos` the test on the member must be positive, in other words, `mtest` on the member succeeds if the test on the member succeeds. In the case of `neg` the test on the member must be negative, in other words, `mtest` on the member succeeds if the test on the member fails. Now, ⟨*test*⟩ represents a predicate with several arguments one of which is `el(`⟨*var*⟩`)`, which is a placeholder for the tested element of the list. So, the test consists of the evaluation of this predicate, in which the argument represented by `el(`⟨*var*⟩`)` is replaced by the element of the list that is currently tested.

EXAMPLE 6.1 Suppose we want to test if a list contains a variable. Besides this test the remainder of the list, excluding the variable we may find in the list, is required as output. Suppose, we also want to be able to find all those variables. This test can be expressed by using `member_list` as follows:

`member_list(all_exist, list(List), mtest(pos, var(el(X))), output(RList)),`

where `var` is a predicate which checks if its argument is a variable. Let `is_var(A,B)` be a predicate that evaluates positively if `A` is equal to variable `B`. Suppose, the test above was intended to find out if an element of the list is equal to the variable `Z`, then we had to replace `var(el(X))` by `is_var(el(X),Z)` in the expression of `member_list`.

The predicate `member_list` does not always appear in this long format. In some particular cases some abbreviated form of the `member_list` predicate is used. For instance, `member_list` is called with three arguments skipping the fourth argument, when the output list is not wanted. For other frequently occurring cases, such as testing if a given element appears in a list or selecting an element from a list, an abbreviated call of `member_list` is used. Both cases can be handled similarly. Its interpretation, i. e. , testing or selecting, only depends on how the element is specified by the user, as a constant or as a variable. Those cases are established by the use of the predicate of the general format `member_list(Kind_of_test,list(L),el(Member))` for which the rule:

```
member_list(Kind_of_test,list(L),el(Member)) :-
    member_list(Kind_of_test,list(L),mtest(pos,=(el(Member), X)))
```

is stated. So, its interpretation depends on the appearence of `Member`, i. e. , instantiated before evaluation or not.

EXAMPLE 6.2 Let `L` be instantiated by some list. Suppose this list is not empty. When evaluating `member_list(unique_exist,list(L),el(a))`, it succeeds only if `a` is a member of `L`.
When evaluating `member_list(all_exist,list(L),el(X))`, `X` will be instantiated, where the instantiation corresponds to some element of the list `L`. By repeatedly backtracking each element of the list is found.

In this particular case, it is also possible to get an output list, i. e. , the list excluding the element that is bound to `Member` after evaluating `member_list`, by using the rule:

```
member_list(Kind_of_test,list(L),el(Member),output(RList)) :-
    member_list(Kind_of_test,list(L),mtest(pos,=(el(Member), X)),
    output(RList)).
```

So, using `member_list` allows us to handle any selection and testing of a member of a list in a uniform manner. This uniformity is one of the great advantages of this predicate. Another advantage is that it adds more meaning to the arguments. In the implementation, presented in the next section, `member_list` has been used frequently for such purposes.

6.2.2 Implementation of the Components of $\mathcal{FICCS}$

In this section, the implementation of the components as described in 6.1 is given. As noted earlier, only that part of the implementation corresponding to the internal behaviour of $\mathcal{FICCS}$ is elaborated. The implementation responsible for the interface of $\mathcal{FICCS}$ to the deductive database management system, the GUI of the system and communication between user and the system are not elaborated in detail here.

6.2.2.1 Implementation of Control in $\mathcal{FICCS}$

The implementation of the component that is responsible for the control in $\mathcal{FICCS}$ is given below.

```
ficcs :-
  install_ficcs,
  message(intro_ficcs),
  load_case.

load_case :-
  load_ddb,
  manage_transaction,
  continue_or_quit.
```

The proper files for running *FICCS* are loaded by the predicate `install_ficcs`. The predicate `message` with argument `intro_ficcs` will generate an opening screen containing some information about *FICCS*. The predicate `load_case` controls the second component, called `load_ddb`, and the third component of *FICCS*, called `manage_transaction`. It controls these two components by the predicate `continue_or_quit` which is responsible for asking the user if a new transaction must be checked, a new deductive database must be loaded or the session must be closed.

6.2.2.2 Implementation for Loading a Deductive Database

The second component, which is represented in the implementation by `load_ddb`, is responsible for loading a proper deductive database.

```
load_ddb :-
  load_extension,
  load_intension,
  load_incrules,
  !.
```

The subgoals of `load_ddb` are responsible for loading the separate parts of the deductive database, i. e., the extensional database, the intensional database and the revised inconsistency rules. The first two subgoals are not elaborated here, because they are rather trivial and depend heavily on the graphical user interface of the system and therefore on the special system predicates, which depend on the specific version of Prolog. However, the third subgoal is explored further. This part is responsible for loading the proper set of revised inconsistency rules. The most interesting part of this subgoal appears when the user does not provide that set. In that case `load_incrules` calls the subgoal `ir_construct`. It will backtrack on the `ir_construct(Incrule)` in order to obtain all revised inconsistency rules belonging to the currently loaded deductive database and it will add these revised inconsistency rules to the deductive database. The implementation of `ir_construct` is given in 6.2.4.

6.2.2.3 Implementation of the Transaction Manager

The third component, which is represented in the implementation by `manage_transaction` is elaborated in more detail. The three subcomponents of the transaction manager described in 6.1.2 are called as follows:

```
manage_transaction :-
  get_transaction(Transaction),
  ir_adjust(Transaction,AdjustedTransaction,AdjustedPutree),
  inconsistency_check(AdjustedTransaction,AdjustedPutree),
  !.
```

The first subcomponent is further decomposed in the implementation by dividing the transaction in facts, rules and inconsistency indicators, each of which are obtained differently.

```
get_transaction(transaction(ChangedFacts,ChangedRules,ChangedIIs)) :-
  get_transaction(facts,ChangedFacts),
  get_transaction(rules,ChangedRules),
  get_transaction(iis,ChangedIIs).

get_transaction(facts,facts(new(Insertions),old(Deletions))) :-
  get_insertions(facts,Insertions),
  get_deletions(facts,Deletions).
get_transaction(rules,rules(new(NewRules),old(RuleIds))) :-
  get_insertions(rules,NewRules),
  get_deletions(rules,RuleIds).
get_transaction(iis,iis(new(NewIIs),old(IIIds))) :-
  get_insertions(iis,NewIIs),
  get_deletions(iis,IIIds).
```

The predicates `get_insertions` and `get_deletions` are calls to input forms which are responsible for obtaining the facts, rules and inconsistency indicators from the user and transforming the input to the internal representation of updates of facts, rules and inconsistency indicators respectivily. The internal representation of insertions and deletions of facts, rules and inconsistency indicators, which are lists, are given in 6.2.1.1. In the database, rules and inconsistency indicators are uniquely identified by using identifiers. Therefore, when deleting rules and/or inconsistency indicators, it is sufficient to specify their identifier. In case of insertions of rules and inconsistency indicators, identifiers are not given by the user, but are provided by the system. This is one of the responsibilities of the predicate `ir_adjust`.

The first argument of the predicate `ir_adjust` contains the current transaction and delivers the adjusted transaction. This adjusted transaction contains, besides the information of the original transaction, the change in the set of revised inconsistency rules. This predicate uses the information produced by the derivation of the revised inconsistency rules to accelerate the adjustment of the set of revised inconsistency rules. Further, some information produced by the adjustment of the revised inconsistency rules is stored in the third argument of `ir_adjust`. The implementation of `ir_adjust` is described in 6.2.4. When the adjusted transaction is derived, then it becomes the input for the inconsistency check, performed by the predicate `inconsistency_check`.

```
inconsistency_check(AdjustedTransaction,_) :-
```

```
  speed(inconsistent(AdjustedTransaction)),
  !.
inconsistency_check(AdjustedTransaction,AdjustedPutree) :-
  message(allowed,AdjustedTransaction),
  commit(AdjustedTransaction,AdjustedPutree),
  nl,
  !.
```

The meta-predicate speed evaluates its argument and delivers as side effect a popup window informing the user about the timing of the evaluation of its argument. In this case, the predicate inconsistent checks if the adjusted transaction leads to an inconsistent state. If it does, no further actions are necessary, because the current database state has not changed yet. Hence, the inconsistency is only signaled and is not repaired somehow. When the evaluation of the goal inconsistent(AdjustedTransaction) does lead to false, the second clause of inconsistency_check is evaluated, which happens when the database is consistent. In this case message informs the user that the transaction is accepted and commit will commit the transaction, when the user has given permission to commit it.

The predicate inconsistent is elaborated as follows:

```
inconsistent(AdjustedTransaction) :-
  selector(Type,AdjustedTransaction,Update),
  ir_apply(Type,Update,IncRule,AdjustedTransaction),
  report(Type,Update,IncRule,AdjustedTransaction,incrules).

selector(fact,adjusted(facts(new(FL),_),_,_,_),ins(F)) :-
  member_list(all_exist,list(FL),el(F)).
selector(fact,adjusted(facts(_,old(FL)),_,_,_),del(F)) :-
  member_list(all_exist,list(FL),el(F)).
selector(rule,adjusted(_,rules(new(RL),_),_,_),ins(R)) :-
  member_list(all_exist,list(RL),el(R)).
selector(rule,adjusted(_,rules(_,old(RL)),_,_),del(R)) :-
  member_list(all_exist,list(RL),el(R)).
selector(ii,adjusted(_,_,iis(new(IIs),_)),ins(II)) :-
  member_list(all_exist,list(IIs),el(II)).

select_incrule_new(adjusted(F,R,II,incrules(new(IncRules),_)),IncRule) :-
  member_list(all_exist,list(IncRules),el(IncRule)).
select_incrule(AdjustedTransaction,incrule(Id,Head,Body)) :-
  incrule(Id,Head,Body),
  \+ is_incrule_old(AdjustedTransaction,Id).
is_incrule_old(adjusted(F,R,II,incrules(_,old(IncRuleIds))),Id) :-
  member_list(unique_exist,list(IncRuleIds),el(Id)).
```

```
ir_apply(Type,Update,incrule(Id,head(Update),body(B)),AdjustedT) :-
  ( select_incrule(AdjustedT,incrule(Id,head(Update),body(B)));
    select_incrule_new(adjustedT,incrule(Id,head(Update),body(B))) ),
  evaluate_body(B,AdjustedT).

evaluate_body([],AdjustedTransaction).
evaluate_body([new(Q)|QL],AdjustedTransaction) :-
  new(Q,AdjustedTransaction),
  evaluate_body(QL,AdjustedTransaction).
evaluate_body([old(Q)|QL],AdjustedTransaction) :-
  old(Q),
  evaluate_body(QL,AdjustedTransaction).
```

The predicate `selector` selects an arbitrary update from the transaction. For each selected update an arbitrary revised inconsistency rule is applied by the predicate `ir_apply`, and the predicate `report` informs the user when the body of the inconsistency rule holds in the updated database. The evaluation of the body of the inconsistency rule is handled by the predicate `evaluate_body`. It calls the query evaluator by evaluating `new` and `old`. Note that when evaluating `new` a second argument is asserted, which contains the adjusted transaction, in order to be able to evaluate a query in the updated database. The report will show the cause of the inconsistency, the rules involved, etc.. The implementation of the predicate `report` is not elaborated, but note that it contains the arguments needed in order to generate a proper report for the user.

The clauses of `ir_apply` represent the algorithm that is needed for checking the consistency of the database. The application of inconsistency rules is not done in a straightforward manner, which would be to apply all updates in the transaction to all related revised inconsistency rules in the updated database. Some strategy is chosen in order to get the most optimal check. This optimization is explained in the next section.

6.2.3 Evaluation of Revised Inconsistency Rules in $\mathcal{FICCS}$

The evaluation of revised inconsistency rules can be performed in several ways. In this implementation the responsibility of the optimization of the queries, which have to be answered in order to perform the inconsistency check, lies with Prolog, while queries to base relations are performed by the database management system. We have created a deductive database system by coupling a relational database system to Prolog. As a result, deductive database rules were given in a special format instead of Prolog rules. Therefore, query evaluation must be explicitly implemented, in order to handle those deductive rules properly. When $\mathcal{FICCS}$ had been integrated in a real deductive database management system, the query evaluator and the rule manager of this database management system could have been used and this section would be superfluous. However, some redundancy in the handling of revised inconsistency rules can better be traced and solved by $\mathcal{FICCS}$, because it belongs to the intrinsic behaviour of revised inconsistency rules. Further, in this section the evaluation of revised inconsistency rules in case of replacements is implemented.

6.2.3.1 Redundancy in the Evaluation of Revised Inconsistency Rules

The obvious way to handle a transaction in *FICCS* is to apply each update in the transaction to the updated set of revised inconsistency rules. However, following this algorithm some redundancy may occur, when facts in the transaction are applicable to rules or inconsistency indicators in the transaction. For instance, suppose a new rule R is inserted into the database. Then the set of revised inconsistency rules IR is updated. Suppose a fact update, say F, also in the transaction, is applicable to one of those updates, say U_{IR}. Let II be the inconsistency indicator that is checked by U_{IR} and let II_F be the revised inconsistency indicator appearing in the body of U_{IR} with respect to some L in II. R has been used in constructing the base revised inconsistency rule U_{IR}. The related revised inconsistency indicator with respect to F, say II_F in the body of U_{IR}, is evaluated when applying F to U_{IR}. However, suppose R is applied to the derived revised inconsistency rule, where the revised inconsistency indicator that subsumes II_F appears in its body. Then the application of the new base U_{IR} with respect to fact F and II is subsumed by the application of that derived revised inconsistency rule with respect to R and II. This case is illustrated by the following example.

EXAMPLE 6.3 Let D be a database consisting of the following facts and rules:

F_1: `patient(11355, broken_leg, 'PARACETAMOL')`

F_2: `disease(broken_leg, fractures)`

F_3: `disease(heartattack, heart)`

F_4: `medicine('PARACETAMOL', 0)`

F_5: `medicine('SOMATONORM', 10)`

F_6: `medicine('DURABOLIN', 10)`

F_7: `medicine('HUMATROPE', 10)`

F_8: `medicine('STOMBA', 15)`

F_9: `location(11355, 'A123')`

```
rule(rule_id(1), head(weak(Pat)), body([patient(Pat, D, Med),
    disease(D, heart)]))
rule(rule_id(2), head(light(Med)), body([medicine(Med, 0)]))
rule(rule_id(3), head(light(Med)), body([medicine(Med, 5)]))
rule(rule_id(4), head(strong(Med)), body([medicine(Med, 10)]))
rule(rule_id(5), head(strong(Med)), body([medicine(Med, 15)]))
```

The database describes a hospital, consisting of the base relations `patient`, `disease`, `medicine` and `location`. The predicate `patient` contains the patient's id number, the reason for admission and the medicine the patient has been prescribed. The predicate `disease` classifies diseases into classes of diseases. The predicate `medicine` classifies medicines into classes of medicines,

which are identified by a number. The predicate `location` states in which hospital room a patient resides. The first rule states that a patient with a disease, which is classified as a heart disease, is a weak patient. The other rules classifies medicines in light and strong ones. Let

```
ii(ii_id(1),[weak(Pat1),location(Pat1,Room),location(Pat2,Room),
  \+ Pat1 = Pat2])
```

be the inconsistency indicator specified for D. It states that a weak patient must have a room for him/herself. Note that D is consistent. The following base and derived inconsistency rules are constructed for D.

```
incrule(incrule_id(0),head(ins(ii(II)),body([new(II)])))
incrule(incrule_id(1),head(ins(patient(Pat1,D,Med))),
  body([new(disease(D,heart)), new([location(Pat1,Room),
  location(Pat2,Room),\+ Pat1 = Pat2])
incrule(incrule_id(2),head(ins(disease(D,heart))),
  body([new(patient(Pat1,D,Med)), new([location(Pat1,Room),
  location(Pat2,Room),\+ Pat1 = Pat2])
incrule(incrule_id(3),head(ins(rule(head(weak(Pat1),body(B))),
  body([new(B), new(location(Pat1,Room), location(Pat2,Room),
        \+ Pat1 = Pat2]))
incrule(incrule_id(4),head(ins(location(Pat1,Room))),
  body([new(weak(Pat1)), new([location(Pat2,Room), \+ Pat1 = Pat2])
incrule(incrule_id(5),head(ins(location(Pat2,Room))),
  body([new(weak(Pat1)), new([location(Pat1,Room), \+ Pat1 = Pat2])
```

Note that only one derived revised inconsistency rule is constructed yet. No derived revised inconsistency rules for the derived predicates `light` and `strong` are constructed because they do not influence any inconsistency indicator. Consider a transaction T consisting of the insertions of the facts

`patient(10134, infection,'DURABOLIN')`

`location(10134,'A123')`

and the rule:

`rule(head(weak(Pat)), body([patient(Pat, D, Med), strong(Med, C)])).`

Then, the new base and derived revised inconsistency rules constructable from the new rule in the transaction are:

```
incrule(incrule_id(6),head(ins(patient(Pat1,D,Med))),
  body([new(strong(Med)), new(location(Pat1,Room),
        location(Pat2,Room), \+ Pat1 = Pat2))])
incrule(incrule_id(7),head(ins(medicine(Med,10))),
  body([new(patient(Pat1,D,Med)), new(location(Pat1,Room),
```

```
        location(Pat2,Room), \+ Pat1 = Pat2]))]
incrule(incrule_id(8),head(ins(medicine(Med,15))),
  body([new(patient(Pat1,D,Med)), new(location(Pat1,Room),
        location(Pat2,Room), \+ Pat1 = Pat2]))]
incrule(incrule_id(9),head(ins(rule(head(strong(Med),body(B))),
  body([new(B),new(patient(Pat1,D,Med)), new(location(Pat1,Room),
        location(Pat2,Room), \+ Pat1 = Pat2]))]
```

Note that, following the algorithm of applying all updates in the transaction to the revised inconsistency rules of the updated database, the base revised inconsistency rules 1, 4, 5 and 6 are applied by the fact updates in T and the derived revised inconsistency rule 3 is applied by the rule update in T.

REMARK Note that `patient(10134, infection,'DURABOLIN')` can be applied to revised inconsistency rule 6, which leads to the evaluation of

```
new(strong('DURABOLIN')), new(location(10134,Room), location(Pat2,Room),
  \+ 10134 = Pat2)).
```

However, the insertion of the rule can be applied to revised inconsistency rule 3, which will lead to the evaluation of

```
new(patient(Pat1,D,Med), strong(Med)), new(location(Pat1,Room),
  location(Pat2,Room), \+ Pat1 = Pat2.
```

This case subsumes the evaluation of revised inconsistency rule 6, when substituting

$$\{\texttt{Pat1/10134, D/infection, Med/'DURABOLIN'}\}$$

as a result of the application of `patient(10134, infection,'DURABOLIN')` to this revised inconsistency rule.

This example shows the redundancy in the application of facts to new revised inconsistency rules, when a fact in the transaction is applicable to a rule in the transaction. Note that a fact update in the transaction is only applicable to an update of the set of base revised inconsistency rules if it is applicable to a rule in the transaction. When the rule update is applied to all relevant derived revised inconsistency rules, the case of the fact update applicable to the updated part of the set of revised inconsistency rules is covered. The same argument holds when applying fact updates to updates of the set of base revised inconsistency rules which are constructed from the updates of inconsistency indicators. The application of those base revised inconsistency rules in the adjusted transaction by facts in the transaction are subsumed by the application of the revised inconsistency rule for the insertion of an inconsistency indicator; for, this revised inconsistency rule will evaluate the full inconsistency indicator.

Therefore, the following algorithm suffices for checking the consistency of the database.

(i) Apply fact updates in the transaction to base revised inconsistency rules that do not appear in the adjusted transaction as inserted or deleted base revised inconsistency rules in the updated database.

(ii) Apply rule updates in the transaction to each of the derived revised inconsistency rules in the updated set of derived revised inconsistency rules in the updated database.

(iii) Check the new inconsistency indicators in the updated database.

REMARK The second step in the algorithm subsumes the application of fact updates to updated base revised inconsistency rules as a result of updates of rules. The third rule in the algorithm subsumes the application of fact updates to updated base revised inconsistency rules as a result of updates of inconsistency indicators.

EXAMPLE 6.4 Consider the database as described in EXAMPLE 6.3. Following this algorithm for the application of revised inconsistency rules, the base revised inconsistency rules 1, 4 and 5 are applied by the fact updates in T and the derived revised inconsistency rule 3 is applied by the rule update in T.

The improved algorithm for the application of revised inconsistency rules is implemented by using clauses `ir_apply`, defined as follows.

```
ir_apply(fact,Update,incrule(Id,head(Update),body(B)),AdjustedT) :-
  select_incrule(AdjustedT,incrule(Id,head(Update),body(B))),
  evaluate_body(B,AdjustedT).
ir_apply(rule,RuleUpdate,incrule(Id,head(RuleUpdate),body(B)),AdjustedT) :-
  ( select_incrule(AdjustedT,incrule(Id,head(RuleUpdate),body(B))) ;
    select_incrule_new(AdjustedT,incrule(Id,head(RuleUpdate),body(B))) ),
  evaluate_body(B,AdjustedT).
ir_apply(ii,IIUpdate,incrule(Id,head(IIUpdate),body(B)),AdjustedT) :-
  incrule(Id,head(IIUpdate),body(B)),
  evaluate_body(B,AdjustedT).
```

6.2.3.2 Implementation of a Query-evaluator

In *FICCS* a special query-evaluator is used, because rules cannot be applied directly since they do not correspond to Prolog-rules. In fact, those rules are presented as general facts. This format was chosen in order to make the management of rules and revised inconsistency rules a lot easier. Besides this advantage, the representation makes the distinction between the rules defining *FICCS* and the rules used in the deductive database explicit.

Further a special purpose query-evaluator is necessary in order to reason in two different database states, i. e. , the current database state and the updated database state. It must also be possible to evaluate inserted rules that appear in the transaction, while deleted rules must be skipped.

At evaluation time, the meta-predicate `new`, used in the implementation of `evaluate_body`, has

two arguments, the first one represents the query, the second argument represents the adjusted transaction. In this form the query evaluator is called. In the case of the evaluation of `old`, it does not require a second argument, because the query evaluator must evaluate the query in the current database state. The implementation of this query evaluator is given below.

```
old(Query) :-
  list(Query),
  empty_transaction(T),
  query(Query,T).
old(Query) :-
  \+ list(Query),
  empty_transaction(T),
  primitive_query(Query,T).

new(Query,T) :-
  list(Query),
  query(Query,T).
new(Query,T) :-
  \+ list(Query),
  primitive_query(Query,T).

query([],T).
query([X|L],T) :-
  primitive_query(X,T),
  query(L,T).

primitive_query(X,T) :-
  X =.. [F|Args],
  query_type(F,Type),
  Type,
  primitive_query(Type,X,T).

query_type(F,compos(F)).
query_type(\+,negated(\+)).
query_type(F,evaluable(F)).
query_type(F,comp(F)).
query_type(F,intensional(F)).
query_type(F,extensional(F)).

primitive_query(compos(F),(X , Y),T) :-
  query([X],T),
  query([Y],T).
primitive_query(compos(F),(X ; Y),T) :-
```

```
  query([X],T);
  query([Y],T).
primitive_query(negated(F),\+ X,T) :-
  \+ query([X],T),
  !.
primitive_query(evaluable(F),X,T) :-
  X,
  !.
primitive_query(comp(F),X,T) :-
  X,
  !.

primitive_query(intensional(F),X,adjusted(F,rules(_,old(RuleIds)),II,IR)) :-
  rule(RuleId,head(X),body(L)),
  \+ member_list(unique_exist,list(RuleIds),el(RuleId)),
  query(L,adjusted(F,rules(new(Rules),old(RuleIds)),II,IR)).
primitive_query(intensional(F),X,adjusted(F,rules(new(Rules),_),II,IR)) :-
  member_list(all_exist,list(Rules),el(rule(RuleId,head(X),body(L)))),
  query(L,adjusted(F,rules(new(Rules),old(RuleIds)),II,IR)).

primitive_query(extensional(F),X,T) :-
  evaluate_once(X,F,OnceOrMore),
  db_once(X,T,T_or_E),
  query_db(T_or_E,OnceOrMore,X,T).

evaluate_once(X,F,once) :-
  prim_key(F,N),
  X =.. [F|Args],
  member_list(num(N),list(Args),out(A1)),
  \+ var(A1),!.
evaluate_once(X,F,once) :-
  \+ prim_key(F,N),
  X =.. [F|Args],
  member_list(all,list(Args),mtest(neg,var)),
  !.
evaluate_once(X,F,more) :-
  !.

db_once(X,adjusted(facts(new(Facts),_),_,_,_),transaction) :-
  only_prim_query(X,Y),
  member_list(all_exist,list(Facts),el(X)),
  !.
db_once(X,T,T_or_E) :-
```

```
  !.

only_prim_query(X,Y) :-
  X =.. [F|Args],
  prim_key(F,N),
  make_duplicate(Args,Args1,N),
  Y =.. [F|Args1],
  !.

query_db(transaction,once,Insertion,adjusted(facts(new(Facts),_),_,_,_)) :-
  member_list(unique_exist,list(Facts),el(Insertion)).
query_db(extensional,once,X,T) :-
  once(X).
query_db(T_or_E,more,X,adjusted(facts(new(Facts),_),_,_,_)) :-
  member_list(all_exist,list(Facts),el(X)).
query_db(T_or_E,more,X,adjusted(facts(_,old(Facts)),_,_,_)) :-
  status_db_call(X),
  X,
  \+ member_list(unique_exist,list(Facts),el(X)).
```

The query evaluator classifies the query in one of the six types, which are described by the predicate `query_type`. The query can be composite, in which case it must first be decomposed before getting a primitive query. This is also applicable to negated queries, which will be decomposed by negating the result of the query. The queries of other query types can be evaluated without any further decomposition. Besides queries on intensional or extensional predicates, some queries do not correspond directly to the database predicates and can be evaluated by Prolog directly, i.e., the evaluable predicates, which are used for computing some result and are defined by Prolog rules, or the comparison predicates such as $<$, $>$, $=$, etc.. The answering of queries containing database predicates is more complex.

Because database accesses are expensive compared to inference steps in the working memory of the system, *FICCS* avoids database accesses as much as possible. Therefore, in order to answer a query in the updated database , three possibilities occur, namely

- ➢ the query can be answered by accessing the transaction, which is stored in main memory,
- ➢ the query can be answered by accessing the old database, which is stored on disk, and
- ➢ the query cannot be answered.

Therefore, the first possibility has a higher priority than the second one, which is implemented by the order of the clauses of `query_db`. Now, when a query has to be evaluated in the updated database, we first look if the query can be answered by the transaction. This may happen regularly, because a fact in a transaction violating some constraint may often be compensated by some other update in the transaction. For instance, in a hospital a patient must have a room. Therefore, when a patient is inserted into the database, the fact representing the room of the patient should

appear in the transaction as well. When a query cannot be answered by the transaction, and it is necessary to look further for an answer, then we try to answer the query by accessing the database on secondary storage. In some cases, we only have to search through the transaction in order to handle the query, avoiding unnecessary accesses of the database.

It is also possible that the database has to be accessed once, avoiding other accesses of the database. Some of these cases are elaborated in this implementation of the query evaluator by the predicates `evaluate_once` and `db_once`. For instance, when a query is ground, then it only has to be evaluated once. If it is possible to answer such a query by the transaction, then even no database access is necessary. When a query consists of a predicate with a primary key, where the value of that key is known at query-time, the query has to be answered only once because there is only one such answer. Hence, no backtracking is needed in order to find other instances. Even when the query cannot be answered, the search process to an answer can be cancelled. Note that when a fact in the database is found for which its key values match the key values of the query, while a complete match is not possible, because the query does not match the fact in some other position of the predicate, then the query should fail instantly; for, the key value uniquely identifies the fact in the database (or the transaction). So, no further search should be performed. Hence, in the case of instantiated primary keys, at most one database access is necessary here. Note that the only condition that has to be fulfilled in order to use this optimization is that the system is responsible for primary key management, i. e. , the system is responsible for assigning unique primary key values to committed updates of a transaction.

REMARK Another type of redundancy is recognized here. The redundancy in the evaluation of queries in which a primary key is involved is called *primary key* redundancy.

6.2.3.3 Implementing Replacements

In 5.2.3 it was showed that replacements can easily be integrated in the method based on revised inconsistency rules. Before replacements are usable in *FICCS*, first the structure of revised inconsistency rules has to be adjusted. Additional information about the variables occurring in a revised inconsistency rule should be available, before we are able to apply a replacement to the revised inconsistency rule. This information, represented by an update variable list, is incorporated into the definition of the revised inconsistency rule as follows:

⟨*incrule*⟩ ≡

`incrule(`⟨*incrule id*⟩`, head(varlist([`⟨*var*⟩*`]), `⟨*update type*⟩`(`⟨*update pattern*⟩`))),`

`body([`⟨*query*⟩*`])).`

Now, the Prolog program handling replacements in this manner can be written as:

```
ir_apply(fact, rpl(Repl), incrule(Id,head(UpdateVarList,Update),body(B)),
                          AdjustedTransaction) :-
  replacement_list(Repl, ReplaceList),
  del_ins(Repl, del(ReplOld), ins(ReplNew)),
```

```
  (Update = del(ReplOld) ; Update = ins(ReplNew)),
  select_incrule(AdjustedTransaction,
                 incrule(Id,head(UpdateVarList,Update),body(B))),
  test_evaluate_body(UpdateVarList, ReplaceList),
  evaluate_body(B, AdjustedTransaction).
```

In this program `replacement_list` determines the replacement list, `del_ins` separates the deletion from the insertion with respect to this replacement (see 5.2.3). Hereafter, these two separated updates are applied to the set of base revised inconsistency rules, following the improved algorithm for the application of revised inconsistency rules. The body of an applicable revised inconsistency rule is only evaluated, when the intersection of the update variable list and the replacement list is not empty. For this purpose a predicate `test_evaluate_body` is introduced in order to test if the body of the revised inconsistency rule has to be evaluated or not. Hence, the Prolog program for defining `test_evaluate_body` can be written as:

```
test_evaluate_body(varlist(UpdateList), varlist(ReplaceList)) :-
  not var(ReplaceList),
  not empty_intersection(UpdateList, ReplaceList).
test_evaluate_body(UpdateList, ReplaceList) :-
  var(ReplaceList).
```

The revised inconsistency rule-generator given in the next section does not incorporate replacements. Hence, it should be adjusted in order to automatically generate revised inconsistency rules containing the update variable list in the head of the inconsistency rules.

Note that we can generalize replacements of facts to replacements of rules and inconsistency indicators. For instance, suppose only one literal of a rule or inconsistency indicator is replaced by another literal, while the remainder of the rule or inconsistency indicator is not changed. We can look at a replacement of a rule (resp. inconsistency indicator) as a deletion and an insertion of the rule (resp. inconsistency indicator) just as the replacement of facts. This also implies that the construction of revised inconsistency rules with respect to such a rule or inconsistency indicator does not have to be done completely from scratch: they have to be adjusted only in those places for which the replacement is relevant. However, these replacements and their consequences for *FICCS* are not elaborated in this thesis and are left as remaining research issues.

6.2.4 Revised Inconsistency Rule Management in *FICCS*

When the set of deductive rules and inconsistency indicators is fixed, the inconsistency rules only have to be generated once and can be used over and over again for updates of facts. This section shows that it is possible to implement a generator for revised inconsistency rules, which generates the set of revised inconsistency rules from the set of rules and indicators. The construction process is divided into the phase of finding a revised inconsistency rule built from the rules and inconsistency indicators and the phase of assigning an identifier to that revised inconsistency rule. All revised inconsistency rules are now found by backtracking. The implementation that controls the construction process is described by the following code.

```
ir_construct(incrule(incrule_id(IncRuleID),H,B),
             putree(incrule_id(IncRuleID),IIId,UsedRules)) :-
  ir_construct(KindofIR,incrule(H,B),putree(IIId,UsedRules)),
  id_manager(incrules,select(IncRuleID)).
```

When a revised inconsistency rule is constructed, some information about the construction process is collected. The predicate represents a part of the potential AND/OR update tree. The predicate `putree` has three arguments. The first argument identifies the revised inconsistency rule, the second argument identifies the inconsistency indicator which is checked by the revised inconsistency rule, and the third argument contains a list in which all rules used for constructing the revised inconsistency rule are collected. This knowledge is used when the set of revised inconsistency rules is adjusted, see the implementation of the predicate `ir_adjust`. The implementation of the predicates `ir_construct`, `id_manager` and `ir_adjust` are elaborated in the remainder of this section, where the part of `ir_construct` for deriving the update expressions containing recursive predicates is distinguished from the the part deriving the update expressions that do not contain recursive predicates.

6.2.4.1 A Revised Inconsistency Rule Generator without Recursion

In 6.2.2.2 a remark was made that the predicate `load_incrules` was responsible for loading the proper revised inconsistency rules when available. However, when the user cannot give those inconsistency rules, they have to be generated from scratch. The revised inconsistency rule generator is implemented as follows.

```
ir_construct(ii,incrule(head( ins( ii(B) )),body(B) ),putree(ii_id(0),[])).
ir_construct(KindofIR,incrule(head(C),body(RevII)),
             putree(IIId,Used_rules)) :-
  revised_ii(H,C1,IIId,RevII1,Used_rules,Ext),
  kind_of_ir(KindofIR,C1,RevII1,C,RevII).

revised_ii(L,C,IIID,RevII,Used_rules,ExtInt) :-
  ii_literal(IIID,L,RemainderII),
  delta(L,C,Collected,Used_rules),
  ext_or_int(C,ExtInt),
  append(Collected,[new(RemainderII)],RevII).

ii_literal(IIID,SignLit,RemainderII) :-
  ii(IIID,II),
  remainder(SignLit,II,RemainderII).

delta([S,H],[S2,A],NewQuery,used_rules([RuleID|RuleIDs])) :-
  directly_depends_on(RuleID,H,[SH,AH],RemainderB),
  \+ recursive(H),
  delta([SH,AH],[S1,A],Rest,used_rules(RuleIDs)),
```

```
  turn_sign(S,S1,S2),
  db_evaluation([S,RemainderB],Query),
  append(Query,Rest,NewQuery).
delta([S,RecLit],[S2,A],NewQuery,used_rules(NewRuleIDs)) :-
  delta_recursive([S,RecLit],[SH,AH],RecQuery,used_rules(RecRuleIDs)),
  delta([SH,AH],[S1,A],Rest,used_rules(RuleIDs)),
  append(RecRuleIDs,RuleIDs,NewRuleIDs),
  append(Rest,RecQuery,NewQuery),
  turn_sign(S,S1,S2).
delta(C,C,[],used_rules([])) :-
  \+ var(C),
  !.

directly_depends_on(RuleID,H,SignLit,RemainderB) :-
  rule(RuleID,head(H),body(B)),
  remainder(SignLit,B,RemainderB).

kind_of_ir(base,[+,A],RevII,ins(A),RevII) :-
  extension([+,A]).
kind_of_ir(base,[-,A],RevII,del(A),RevII) :-
  extension([-,A]).
kind_of_ir(derived,[+,A],RevII,ins(rule(head(A),body(B))),[new(B)|RevII]) :-
  intension([+,A]).
kind_of_ir(derived,[-,A],RevII,del(rule(head(A),body(B))),[old(B)|RevII]) :-
  intension([-,A]).

%-------------------------------------------------
% auxiliary predicates

remainder(SignLit,B,RemainderB) :-
  member_list(all_exist,list(B),el(Lit),output(RemainderB)),
  match(Lit,SignLit,+),
  (extension(SignLit);
   intension(SignLit)).

db_evaluation([S,[]],[]).
db_evaluation([-,RemainderB],[old(RemainderB)]) :-
  \+ RemainderB = [].
db_evaluation([+,RemainderB],[new(RemainderB)]) :-
  \+ RemainderB = [].

recursive(H) :-
  recursive(H,_,_),
```

```
  !.
recursive(H,rule(ID,head(H),body(B)),body(H1,Remainder)) :-
  H =.. [F|_],
  rule(ID,head(H),body(B)),
  member_list(all_exist,list(B),el(H1),output(Remainder)),
  H1 =.. [F|_],
  !.

intension([S,H]) :-
  rule(_,head(H),_),!.

intensional(F) :-
  rule(_,head(H),_),
  H =.. [F|_],
  !.

extension([S,A]) :-
  A =.. [F|_],
  extensional(F),!.

ext_or_int(SignLit,extensional) :-
  extension(SignLit),
  !.
ext_or_int(SignLit,intensional) :-
  !.

turn_sign(+,+,+).
turn_sign(+,-,-).
turn_sign(-,+,-).
turn_sign(-,-,+).
```

6.2.4.2 A Revised Inconsistency Rule Generator with Recursion

When recursion is introduced in rules leading to an inconsistency indicator, the implementation has to be adjusted. The predicate `delta`, which is responsible for the construction of update expressions, must be enabled to handle recursion as well. This is accomplished by adding the following clause to the clauses of `delta` in the implementation.

```
delta([S,RecLit],[S2,A],NewQuery,used_rules(NewRuleIDs)) :-
  delta_recursive([S,RecLit],[SH,AH],RecQuery,used_rules(RecRuleIDs)),
  delta([SH,AH],[S1,A],Rest,used_rules(RuleIDs)),
  append(RecRuleIDs,RuleIDs,NewRuleIDs),
  append(Rest,RecQuery,NewQuery),
```

```
  turn_sign(S,S1,S2).
```

It uses the predicate `delta_recursive` in order to derive the proper update expressions, which is implemented by:

```
delta_recursive([S,RecLit],[SH,AH],NewQuery,UsedRules) :-
  tc_initialize(UsedRules,RecLit,TC,B2,YList),
  delta_tc([S,RecLit],TC,RecQuery,B2,[SH,AH],Remainder,UsedRules,YList),
  db_evaluation([S,Remainder],Query),
  append(Query,RecQuery,NewQuery).

tc_initialize(used_rules([RuleID1,RuleID2]),RecLit,TC,B2,YList) :-
  recursive(RecLit,rule(RuleID1,head(RecLit),body(B1)),body(BRecLit,TC)),
  rule(RuleID2,head(RecLit),body(B2)),
  \+ RuleID2 = RuleID1,
  list_vars(RecLit,BRecLit,XList,ZList,YList),
  assert(linrec(used_rules([RuleID1,RuleID2]),TC,XList,ZList)).

list_vars(H,H1,XList,ZList,YList) :-
  H =.. [R|HArgs],
  H1 =.. [R|BArgs],
  separate(HArgs,BArgs,XList,ZList,YList).

separate([],[],[],[],[]).
separate([X|HArgs],[Z|BArgs],[X|XList],[Z|Zlist],YList) :-
  \+ X == Z,
  separate(HArgs,BArgs,XList,Zlist,YList).
separate([X|HArgs],[Z|BArgs],XList,ZList,[X|YList]) :-
  X == Z,
  separate(HArgs,BArgs,XList,ZList,YList).

delta_tc([S,RecLit],TC,NewQuery,B2,[SH,AH],Remainder,UsedRules,YList) :-
  \+ TC = B2,
  linrec(UsedRules,TC,XList,ZList),
  delta_query1(UsedRules,RecQuery,XList,ZList,TC1),
  remainder([SH,AH],TC1,Remainder),
  db_evaluation([S,B2],Query),
  append(Query,RecQuery,NewQuery).
delta_tc([S,RecLit],TC,RecQuery,B2,[SH,AH],Remainder,UsedRules,YList) :-
  \+ TC = B2,
  linrec(UsedRules,TC,XList,ZList),
  delta_query2(UsedRules,RecQuery,XList,ZList,TC1),
  remainder([SH,AH],B2,Remainder),
  db_evaluation([S,Remainder],Query),
```

```
  append(Query,RecQuery,NewQuery).
delta_tc([S,RecLit],TC,NewQuery,B2,[SH,AH],Remainder,UsedRules,YList) :-
  TC = B2,
  linrec(UsedRules,TC,XList,ZList),
  delta_query3(UsedRules,RecLit,RecQuery,XList,YList,TC1),
  remainder([SH,AH],TC1,Remainder),
  db_evaluation([S,B2],Query),
  append(Query,RecQuery,NewQuery).

delta_query1(UsedRules,TCQuery,XList,ZList,TCUpdate) :-
  create_duplicate_varlist(XList,VList),
  create_duplicate_varlist(ZList,WList),
  linrec(UsedRules,TCUpdate,VList,WList),
  linrec(UsedRules,TC1,XList,VList),
  linrec(UsedRules,TC2,WList,ZList),
  TCQuery = [new([(tc(linrec(UsedRules,TC1,XList,VList));
                   equal_list(XList,VList))]),
             new([(tc(linrec(UsedRules,TC2,WList,ZList));
                   equal_list(ZList,WList))])].

delta_query2(UsedRules,TCQuery,XList,ZList,TCUpdate) :-
  linrec(UsedRules,TC,XList,ZList),
  TCQuery = [new([(tc(linrec(UsedRules,TC,XList,ZList));
                   equal_list(ZList,XList))])].

delta_query3(UsedRules,RecLit,TCQuery,XList,YList,TCUpdate) :-
  create_duplicate_varlist(XList,VList),
  create_duplicate_varlist(YList,WList),
  linrec(UsedRules,TCUpdate,VList,WList),
  RecLit =.. [R|Args],
  append(XList,VList,List1),
  append(WList,YList,List2),
  TC1 =.. [R|List1],
  TC2 =.. [R|List2],
  TCQuery = [new([(TC1;equal_list(XList,VList))]),
             new([(TC2;equal_list(YList,WList))])].
```

The intention of the predicates `tc_initialize` and `delta_tc` is to construct the transitive closure of the recursive predicate. The predicate `delta_tc` distinguishes the three cases of update expressions with respect to predicates appearing in linear recursive definitions, i. e. , the update expression concerning a predicate in the non-recursive part of the linear recursive definition, the update expression concerning a predicate in the recursive part of the linear recursive definition, where in both cases the non-recursive part of the definition does not match the recursive part of

the definition, and the update expression concerning a predicate appearing in the definition, where the non-recursive part of the definition matches the recursive part. These cases are covered by the predicates `delta_query1`, `delta_query2` and `delta_query3` respectivily. The general definition of the transitive closure of the non-recursive part of the recursive rule is implemented by:

```
tc(linrec(TC,VList,WList),T) :-
  new(TC,T).
tc(linrec(TC,VList,WList),T) :-
  create_duplicate_varlist(VList,VNewList),
  linrec(TC1,VNewList,WList),
  new(TC1,T),
  linrec(TC2,VList,VNewList),
  tc(linrec(TC2,VList,VNewList),T).

create_duplicate_varlist(XList,ZList) :-
  number_list(XList,M),
  makelistvar(M,ZList).

number_list([],0) :-
  !.
number_list([X|L],N) :-
  number_list(L,N1),
  N is N1 + 1,
  !.

makelistvar(1,[X]) :-
  !.
makelistvar(M,[X|VarList]) :-
  M1 is M - 1,
  makelistvar(M1,VarList),
  !.
```

6.2.4.3 An Identifier Manager

The identifier manager is responsible for the assignment of identifiers to rules, inconsistency indicators and revised inconsistency rules. When a new rule, inconsistency indicator or revised inconsistency rule is introduced, an identifier is selected from the corresponding identifier list. When a rule, inconsistency indicator or revised inconsistency rule is deleted, its identifier must be made available again by inserting it to the proper identifier list. The identifier manager uses identifier lists of the following format:

$\langle id\ list \rangle \equiv$

`list_of_ids(`$\langle kind\ of\ ids \rangle$`,[`$\langle ID \rangle^*$`]),`

where

⟨*kind of ids*⟩ ≡

```
incrules | rules | iis
```

and [⟨*ID*⟩*]) represents a list of available identifiers. The last member of the identifier list is not the last available number, but states that all numbers beyond it are available too. For instance, the identifier list [5, 8, 12] states that all integers greater than 12 are available as identifiers too. The identifier manager always selects the first identifier from a list. The identifiers in an identifier list are arranged in ascending order.

```
id_manager(Kind_of_id,IdOperation) :-
  retract(list_of_ids(KindofId,IdList)),
  manage_id(IdList,IdOperation,IdListNew),
  assert(list_of_ids(KindofId,IdListNew)),
  !.

manage_id([ID],select(ID),[ID1]) :-
  ID1 is ID + 1,
  !.
manage_id([ID|IdList],select(ID),IdList) :-
  \+ IdList = [],
  !.
manage_id([ID1],unselect(ID),[ID]) :-
  ID1 is ID + 1,
  !.
manage_id(IdList,unselect(ID),IdList1) :-
  ordered_id_list(ID,IdList,IdList2),
  optimize_id_list(IdList2,IdList1),
  !.

ordered_id_list(ID,[N|IdList],[N|IdList1]) :-
  ID > N,
  ordered_id_list(ID,IdList,IdList1),
  !.
ordered_id_list(ID,IdList,[ID|IdList]) :-
  !.

optimize_id_list([ID|IdList],[ID|IdList1]) :-
  \+ one_step_list([ID|IdList]),
  optimize_id_list(IdList,IdList1),
  !.
optimize_id_list([ID|IdList],[ID]) :-
  one_step_list([ID|IdList]),
  !.
```

```
one_step_list([ID]) :-
  !.
one_step_list([ID,ID1|IdList]) :-
  ID is ID1 - 1,
  one_step_list([ID1|IdList]),
  !.
```

Some other identifier manager may be used, such as the one presented in [Kum92], but the presented identifier suffices for this implementation.

6.2.4.4 Adjustment of the Set of Revised Inconsistency Rules

In this section, a description of the implementation of the predicate `ir_adjust` is given. It is responsible for the adjustment of the set of revised inconsistency rules after a transaction containing rule or inconsistency indicator updates. Here, we suppose that the set of revised inconsistency rules is already constructed for the current database state at an earlier stage.

In the construction process of revised inconsistency rules some information about that construction process is stored by the predicate `putree`, see 6.2.4. This predicate stores information about the rules and the inconsistency indicator that are involved in constructing the revised inconsistency rules. Those predicates are generated during the construction process of revised inconsistency rules. In order to adjust the set of revised inconsistency rules after a transaction, this information allows us to make the adjustment of the set of revised inconsistency rules more efficient. The format of the predicate `putree` is as follows:

$\langle putree \rangle \equiv$

`putree(`$\langle incrule\ id \rangle, \langle ii\ id \rangle, [\langle rule\ id \rangle^*]$`).`

The first argument identifies the inconsistency rule that is checked by the revised inconsistency rule. The second one identifies the inconsistency indicator that is involved when the revised inconsistency rule is applied. The third one identifies the rules that were involved in the construction of the revised inconsistency rule.

Note that updating the revised inconsistency rules does not mean that all inconsistency rules have to be generated from scratch. Only the insertions and/or deletions of the revised inconsistency rules that correspond to the rule and inconsistency indicator updates are needed. So, the set of inconsistency rules can be updated incrementally.

For instance, suppose an inconsistency indicator with identifier ID is deleted. Now, the revised inconsistency rules for which their identifier appears in the first argument and ID appears in the second argument of a `putree`-fact should be deleted also. When a rule is deleted, the same reasoning holds for revised inconsistency rules for which the identifier of the deleted rule appears in the list of rule identifiers appearing in the third argument of a `putree`-fact.

When a new inconsistency indicator is introduced, we have to construct the related revised inconsistency rules from scratch. In order to construct these inconsistency rules, the revised inconsistency rule generator of 6.2.4 can be used. In case of an insertion of a rule this generator can also be used in order to construct all revised inconsistency rules and selecting only those inconsistency rules that contain this rule in their rule identifier list. Note that `ir_adjust` can be implemented in this naive way. However, after having dealt with deletions of rules and inconsistency indicators and insertions of inconsistency indicators, this is not the most efficient algorithm in order to determine revised inconsistency rules in case of rule insertions. Let `ins(rule(head(H), body(B))` be a rule insertion. Note that the derivation of new derived revised inconsistency rules with respect to `H` is already covered when the updates of inconsistency indicators were handled in the first place. So, when determining the base revised inconsistency rules with respect to base relations which influence `H`, then some derived revised inconsistency rules with respect to rules with `H` as head are already derived. However, this may not be the case for derived relations in body literals of the inserted rule, which can lead to an inconsistency indicator belonging to the unchanged set of inconsistency indicators, for which no derived revised inconsistency rule exists yet. So, the inserted rule establishes a new connection between body literal and inconsistency indicator. Therefore, the following algorithm excluding the newly inserted inconsistency indicators is applied in order to find the new revised inconsistency rules with respect to the unchanged set of inconsistency indicators.

(i) Determine the update expressions with respect to `H` and a literal L of an inconsistency indicator II that is influenced by `H`, using the information stored in the relation of `putree` of the derived revised inconsistency rule with respect to `H` and II.

(ii) Determine the update expressions of the inserted rule with respect to `H` and B_j for each $j = 1, 2, \ldots, n$.

(iii) If B_j corresponds to a derived relation,

- ➢ if B_j does not influence L in II in the old state, i. e., no derived revised inconsistency rules exists with respect to B_j, L and II, then derive one with respect to B_j and II via L.
- ➢ determine the update expressions with respect to B_j and each base relation influencing B_j for each $j = 1, 2, \ldots, n$, where for each literal through which B_j is reached from this base relation corresponding to a derived relation for which no derived revised inconsistency rule exist, a derived revised inconsistency rule is derived.

(iv) Determine base revised inconsistency rules by composing the update expressions, which are derived in the previous steps, with respect to L and each base relation found in step two or three.

Here B_j represents some body literal of the inserted rule. In this proces the relevant identifiers of rules and/or inconsistency indicators are collected and stored in the predicate `putree`.

EXAMPLE 6.5 Consider the revised inconsistency rules of EXAMPLE 6.3. The following predicates of `putree` were derived during the construction process of the revised inconsistency rules with identifiers `incrule_id(0)`, ... , `incrule_id(4)` at an earlier stage:

putree(incrule_id(0), _, []),

putree(incrule_id(1), ii_id(1), [rule_id(1)]),

putree(incrule_id(2), ii_id(1), [rule_id(1)]),

putree(incrule_id(3), ii_id(1), []),

putree(incrule_id(4), ii_id(1), []).

putree(incrule_id(5), ii_id(1), []).

One can verify that following this algorithm the updated set of revised inconsistency rules (see EXAMPLE 6.3) is derived.

REMARK The derived revised inconsistency rule with identifier incrule_id(9) was derived, before computing the base revised inconsistency rules following the algorithm above.

Let rule_id(6) be the identifier assigned to the rule in the transaction. Note that no inconsistency indicators are present as updates. Therefore, the stage in which new revised inconsistency rules are computed from new inconsistency indicators does not exist. The revised inconsistency rules are derived by looking at the head of the inserted rule first. Note that its head is weak(Pat). Then we look if this literal already leads to the specified inconsistency indicator, in which case we proceed with the algorithm in order to find the remaining inconsistency indicators. The rule identifiers for the rules, through which the inconsistency indicator is derived, are stored; they contain at least the rule identifier of the rule update. Note that weak(Pat) directly influences the inconsistency indicator with ii_id(1); so, only the identifier rule_id(6) is stored as relevant rule identifier yet. From this point, the derived and base revised inconsistency rules are constructed. When for this head literal no derived revised inconsistency rule exists, one is constructed for this head literal, storing the proper information in the predicate of putree. Constructing the base revised inconsistency rules corresponding to the rule update for this point implies the determination of the update expressions of each of the body literals and each base relation of which it depends. Note that in this construction process only rules are applied that do appear in the updated database. When adjusting the set of revised inconsistency rules by this algorithm, the following predicates concerning putree are derived:

putree(incrule_id(6), ii_id(1), [rule_id(6)]),

putree(incrule_id(7), ii_id(1), [rule_id(4), rule_id(6)]),

putree(incrule_id(8), ii_id(1), [rule_id(5), rule_id(6)]),

putree(incrule_id(9), ii_id(1), [rule_id(6)]),

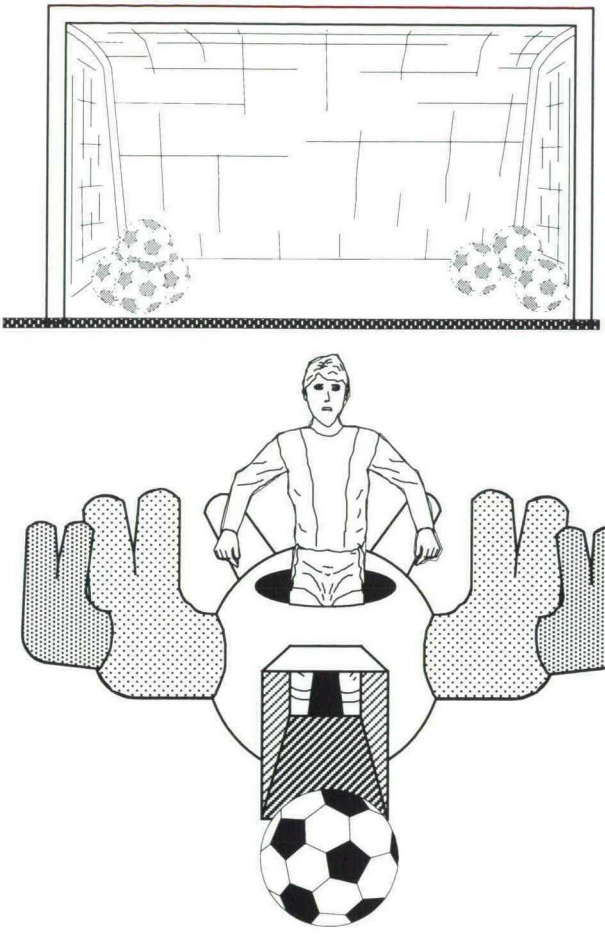
In this example only a rule update appeared. However, an inconsistency indicator update may also appear. Suppose a new inconsistency indicator is inserted into the database, then we have to construct all revised inconsistency rules from this new inconsistency indicator, by making use of all rules in the updated database. When in the same transaction an insertion of a rule appears, the construction of new revised inconsistency rules is constrained to the revised inconsistency rules not corresponding to the new inconsistency indicator, because those revised inconsistency rules

are covered by the revised inconsistency rules constructed for this inconsistency indicator. The implementation of `ir_adjust` follows the algorithm and considerations in the construction of the adjusted set of revised inconsistency rules presented in this section.

References

[Kum92] AKHIL KUMAR. Techniques for Indexing Large Numbers of Constraints and Rules in a Database System. In A. M. TJOA AND I. RAMOS, editors, *Proceedings of the International Conference on Database and Expert Systems Applications*, pages 65–71, Valencia, Spain, 1992. Springer-Verlag.

"In this thesis, inconsistency rules are applied to the update in order to determine in an early stage if it leads to an inconsistency."



Chapter 7

Related Research

Collecting the papers on the subject of integrity checking in deductive databases is not easy, because the contributions are spread along a wide range of proceedings, books, reports and theses. This chapter dedicated to contributions to integrity checking in deductive databases will give an almost complete inventory of available papers on this subject.

7.1 Integrity Checking in Deductive Databases

The idea of simplifying constraint checking, by the assumption that a database is consistent before the update, was originated by Nicolas (see [Nic79]) and Blaustein (see [Bla90]). However, they studied integrity constraint checking in relational databases. In [NY78] the issue of integrity checking in deductive databases was introduced. After those preliminary contributions, several contributions presenting methods for integrity checking in deductive databases were made. Each of the referenced methods in this section assumes that the deductive database is consistent before any transaction, although there exist papers which deal with reasoning with inconsistencies. In [BS89], [GS95] and [MSW91] one reasons with inconsistencies in deductive databases, using paraconsistent logics, non-monotonicity and petri-nets respectively. This section does not pretend to give a full description of the available methods, but only gives some major characteristics of those methods that seem interesting in the perspective of this thesis. The presented methods are roughly classified in methods based on induced updates resp. potential updates, methods which adjust the proof procedure in order to be able to reason forward from the update, methods based on meta-logic programming techniques and methods adjusting the intensional part of the deductive database. More details of each of these methods can be found in the papers to which is referred.

7.1.1 Methods based on Induced Updates

Miyachi et al. One of the first papers on integrity checking in deductive databases is the paper of Miyachi et al., see [MKK+84]. They call their method a *knowledge assimilation method* and consider four types of checks that could be performed. The following sequence of checks is proposed. First, there is a provability check in order to find out if the transaction contains information already present in the current database state, in fact an effectiveness check, in which case the new

database state is equal to the current database state. Secondly, there is a check for contradiction to find out if the transaction is consistent with the current database. In case of an inconsistency the transaction is not performed at all. Thirdly, there is a redundancy check in order to find out if some information I in some part of the database can be derived from some other part of the database together with the transaction, in which case I is redundant and left out of the new database. This redundancy is caused by the kind of database they assume, namely not structured databases. Finally, a so called *independency check* is performed in order to find out if the transaction does influence the consistency of the database derived after the third check. If not, there is no objection for executing the transaction immediately. In fact, the so called *contradiction check* is responsible for checking the integrity constraints, while the provability check is responsible in order to avoid this check. The last two checks are responsible for the execution of the transaction, delivering a database without any redundancy.

In fact, Miyachi et al. presented a method which corresponds to the method based on induced updates. In their method no strategy for handling redundancy is offered. So, redundancies of the first and the third kind, like in the case of methods based on induced updates, are still present. They use Prolog rules for deriving induced updates. Even no duplicate test in the derivation of induced updates is given. However, their attempt to implement integrity checking in deductive databases was quite new at that time.

Decker In [Dec87] a method for integrity checking in deductive databases is presented, which is related to the method based on induced updates. It is a generalization from the relational case to the deductive case of the method of Nicolas in [Nic82]. First, the range-restricted formulas in [Nic82] are generalized. These generalized range-restricted formulas may also contain existential quantors and are transformed to a, so called, *range form*. When the computation rule is adjusted to handle these range form expressions, these range form formulas are expressible in Prolog and also resolvable. This means that from two range expressions another one is resolved.

In the method of [Dec87] effective induced updates are computed instead of ordinary induced updates like in the method based on induced updates. After the computation of an effective induced update, the affected constraints are evaluated. Note that first induced updates are defined by using a more general type of formulas, namely, range expressions.

Martens and Bruynooghe In [MB86] the evaluation of integrity constraints in deductive databases is accomplished by using rule/goal graphs. They extend the rule/goal graphs of Ullman, see [Ull85], which were used for the efficient evaluation of queries in deductive databases without negation, to rule/goal graphs that incorporate integrity constraints and negation in order to optimize evaluation of integrity constraints. These rule/goal graphs capture the information of bound and free variables in queries and rules leading to a literal in the constraint, which is shown by Ullman to be important information to be able to evaluate queries efficiently. In fact, they relate integrity checking with query optimization. They built their findings on the method of Decker, see [Dec87], by the derivation of induced updates. They suggest an effectiveness test

on the updates as well as a redundancy test on the induced updates. Because they do not make a strict distinction between base predicates and derived predicates, i. e. , the database is not structured, an update can redundantly appear as an induced update as well. First, integrity checking is looked at from a function-free perspective. Later, some problems, when incorporating functions in methods for preserving the consistency of deductive databases, are studied. Further, Martens and Bruynooghe allow rules and integrity constraints in transactions, which implies an adjustment of their rule/goal graphs. The contribution of Martens and Bruinooghe in their paper to the field of integrity checking is the use of query optimization techniques in methods for checking the consistency of the database (see [Hum92]).

7.1.2 Methods based on Potential Updates

Lloyd, Sonenberg and Topor In [LT85], [LT86] and [LST87] a method is presented that corresponds to a large extent to the method based on potential updates. Therefore, we refer to CHAPTER 2 for further details of this method.

Bry, Manthey et al. In [BDM87], [BD88], [BM86] and [Bry87] another variant on the method of potential updates is presented. Instead of evaluating the potential instances of the constraints directly, first the potential update is evaluated in order to see if this potential update has an instance which is an effective induced update and which is not a duplicate. For each of those induced updates the instantiated remainder of the constraint is evaluated.

In [BMM91] an overview is given of the methods about integrity checking in the late seventies and the eighties.

Jeusfeld and Krüger Jeusfeld and Jarke show in [JJ91] that checking constraints in Object-Oriented Systems and in relational systems are somewhat alike. Jeusfeld and Krüger show in [JK90] that integrity constraint checking techniques in deductive databases can be applied in an object-oriented environment. They use the integrity checking method for deductive databases introduced by [BDM87]. They argue that redundancy by replacement can be avoided in an object-oriented setting using the features, such as specialisation of objects and methods, offered by the object-oriented data model. However, in this thesis it is shown that in the deductive setting this is also avoidable. They implemented this idea in the knowledge base management system ConceptBase. The specific feature of the presented method is that rules as well as integrity constraints are simplified in order to derive implicit facts, only if it is relevant to some integrity constraint. The presented work in object-oriented databases has a strong resemblance to the method based on inconsistency rules (not revised) in deductive databases.

Das and Williams In [DW89a], [Das90] and [Das92] Das and Williams present their *path finding method*. Integrity constraints are transformed into denials coupled to constraint identifications, i. e. , a constraint has the form $IC(Id) \longleftarrow B$. This method is a mixture of the method

based on induced updates and the method based on potential updates. It has a bottom-up generation phase, in which derived updates are computed. The sign of the derived literal decides whether the derived update is interpreted as an induced update or as a potential update. If the derived update is positive, then we determine all induced updates corresponding to this derived update. However, if the derived update is negative, then we proceed with the potential deletion. In other words, in case of insertions the method corresponds to the method based on induced updates, while in case of deletions this method corresponds to the method based on potential updates.

This means that this method also contains redundancy of the first kind, because irrelevant induced updates may be derived. Also redundancy of the second and third kind may be present in the case of potential deletions.

In [DW89b] Das and Williams use some examples in order to compare and test several methods, namely those of [LST87], [Dec87] and [SK88] and the path finding method of Das and Williams. In the examples of Das and Williams (see [DW89a]) the use of recursive relations is permitted in rules but avoided in the inconsistency indicators itself. In that way, a full check of inconsistency indicators with recursion is avoided. In [Das91] a meta-logic approach is presented.

Asirelli et al. In the work of Asirelli et al. about integrity checking SLDNF-resolution is used as the basic query evaluation technique for answering queries (see [ABI89, AIM88, AdSM85]).

In [AIM88] a method is proposed, which is a combination of the methods based on induced updates and on potential updates. In this method, an instantiation of a potential update is stored before proceeding to the derivation of other potential updates with respect to this potential update. In order to illustrate this, consider the following example.

EXAMPLE 7.1 Consider a deductive database D which contains the rule:

$$a(X, Y, Z) \longleftarrow b(X, Y), c(Y, Z, W), d(X, W).$$

Let $a(X, Y, Z)$ be relevant to an inconsistency indicator. Suppose that $c(1, 2, 3)$ is an update to D. Then the potential update $a(X, 1, 2)$ is derived. Because $a(X, Y, Z)$ appears in the inconsistency indicator, in the method based on potential updates $a(X, 1, 2)$ is evaluated. However, this leads to the evaluation of the instantiated body of the rule:

$$b(X, 1), c(1, 2, W), d(X, W)$$

for which only the instance

$$b(X, 1), c(1, 2, 3), d(X, 3)$$

is relevant for the inconsistency check. In [AIM88] instantiations will be taken into account when deriving potential updates. Instead of the potential update only the rest part of the body of the rule from which it is derived is stored. So, in this example it is expressed by $[a(X, 1, 2), b(X, 1), d(X, 3)]$. This expression is used, when $a(X, 1, 2)$ is evaluated.

Note that in this case redundancy of the second type has been reduced compared to the pure variant of the method based on potential updates. This method would correspond exactly to the method based on induced updates, when the rest part of the potential update expression is evaluated first, before deriving new potential updates.

In [AdSM85] two approaches for handling integrity constraints are distinguished. One that concerns the checking of the constraints, which is used in this thesis. They used SLD-resolution to perform this task only for definite deductive databases. The other concerns how a given deductive database, which is not necessarily consistent with the specified set of constraints, can be modified to get a deductive database corresponding to a minimal model satisfying the set of constraints. This is a strong kind of integrity enforcement, which states that the inconsistency is not caused by the constraints.

Celma, Casamayor, Decker et al. In [CGMD94] the integrity checking methods presented in [BDM87], [DW89a], [LST87] and [SK88] are compared. These methods are described by a meta-level logic language. In this language, one can express how and when the bottom-up generation phase and the top-down evaluation phase, described in [CM92], in each method is handled. This comparison has lead to a new approach, which combines the strong points of the methods presented in [LST87] and [SK88]. This method, called the convergence method, is expressed in this meta-level logic language as well and is a variant of the method based on potential updates, where some bottom-up information in the derivation of potential updates is added to the potential updates. In [CCM+91] this principle is expressed by representing the adjusted potential updates as a triple (P, CA, CB) where P represents the potential update, CA represents some condition that has to be evaluated after the update (i. e. , in the updated database) and CB represents some condition that has to be evaluated before the update (i. e. , in the current database). CA and CB are expressions comparable to the update expressions in the method based on revised inconsistency rules. Because all integrity constraints are represented as rules defining some constant representing an inconsistent state, the conditions of the derived triples of which the first argument is equal to this constant symbol have to be evaluated. An implementation of the proposed method is given in this paper. [CCD93] contributes also to this way of integrity checking, showing that the instantiations derived in generating the potential updates in the method of [AIM88] can be specialised even more. These papers have led to a new method based on adjustments of the proof procedure, which is globally described in the next section.

Moerkotte et al. In [MK88] Moerkotte and Karl discussed the methods of [SK88] and l [LT86] in case of deletions. In their database only Horn clauses are allowed. They gave an example which shows that in some cases the method of [SK88] is less efficient than the method of [LT86]. This difference in efficiency is a typical case of, what is called in this thesis, redundancy of the first type. They gave also an example which shows that in some cases the method of [LT86] is less efficient than the method of [SK88]. This difference in efficiency is a typical case of, what is called in this thesis, redundancy of the second type.

They propose an integrity checking method which combines the methods of [SK88] and [LT86]. On the basis of some statistical knowledge about the extensional database, they decide to reason forward either with effective induced updates or potential updates. They argue that it is in some cases better to instantiate the potential updates, as in the method of Lloyd and Topor, partially. Hereafter, the instantiated potential updates are used as input for the method of Sadri and Kowalski. In other cases it is better to derive induced updates, as in the methods of Sadri and Kowalski. Moerkotte and Karl show the gain of efficiency in their method compared to these methods. To clarify the possible gain of this statistical knowledge in integrity checking an example of Karl and Moerkotte is used.

EXAMPLE 7.2 Let D be a deductive database, for which one inconsistency indicator is specified, with the following rule and fact base:

RULES

R_1: $r(X, Z) \longleftarrow p(X, Y, Z), q(X, Y)$

FACT

F_1: $q(a, b)$

F_2: $p(a, b, c_i)$, for each $i = 1, 2, \ldots, 100$

F_3: $r(a, c_{2j-1})$, for each $j = 1, 2, \ldots, 50$

F_4: $s(a, c_{50})$

INCONSISTENCY INDICATOR

II_1: $\exists X \exists Z[s(X, Z), \neg r(X, Z)]$

It is obvious that this inconsistency indicator does not hold in D, because there is only one substitution in X and Z which makes $s(X, Z)$ true, i. e., $\{X/a, Z/c_{50}\}$, for which also $r(a, c_{50})$ holds because we can apply the rule. Suppose we delete $q(a, b)$ from D. When we try to find effective induced updates first, we find that $r(a, c_{2l})$ can no longer be derived for each $l = 1, 2, \ldots, 50$. Now, each implicit deletion of $r(a, c_{2l})$ will affect the inconsistency indicator, resulting in the repeated evaluation of $s(a, Z)$ for Z/c_{2l}. Only the evaluation of $s(a, c_{50})$ will prove an inconsistency. So, the number of database accesses may be enormous when reasoning from the enormous number of instances of relation r. When we know that r contains great number of facts and that s contains just one fact, we decide to reason forward from the potential update $\neg r(a, Z)$ instead of instantiating it first. After deriving the potential instance of II_1, i. e., $\exists Z[s(a, Z), \neg r(a, Z)]$, one can decide, on the basis of the statistical knowledge of s and r, to evaluate $s(a, Z)$ first. Now, in one step, we find $s(a, c_{50})$ to be true, for which we only have to evaluate one instance of $\neg r(a, Z)$, namely $\neg r(a, c_{50})$, which happens to hold in $D_{q(a,b)}$

When considering the method based on inconsistency rules in this case, we find one inconsistency rule for the deletion of $q(A, B)$-facts, namely

$$inconsistent(del(q(X, Y)) \Leftarrow old(p(X, Y, Z)), new(\neg r(X, Z)), new(s(X, Z)).$$

$\neg r(X, Z)$ is present in the body of the inconsistency indicator, because in this example the distinction between base and derived predicates is dropped, which means that we interpret facts in r as rules with an empty body. The statistical knowledge of relations can be used in this method too, i. e. , we could decide to evaluate $new(s(X, Z))$ before any other call to *query*. This could be expressed by rewriting the inconsistency rule to:

$$inconsistent(del(q(X, Y)) \Leftarrow new(s(X, Z)), old(p(X, Y, Z)), new(\neg r(X, Z)).$$

However, this subgoal ordering is a responsibility of the query evaluator. Moerkotte and Karl try to integrate query optimization in their method, while it should be two different things. Their idea was induced by their choice to implement their method in a Prolog like system, which is, as we have concluded in CHAPTER 1, certainly not equal to a deductive database system. The title of their paper, "Efficient Consistency Control in Deductive Databases", is therefore misleading, because they present a method for efficient consistency control in a prolog system. The method based on inconsistency rules allows the shift of query optimization from the method to the system.

In [ML91] Moerkotte and Lockemann present an experimental integrity maintenance system. The system and the user must cooperate to maintain the consistency of the external consistency. They distinguish three steps in the maintenance process:

(i) find all induced updates and base and derived facts responsible for each violation of a constraint, i. e. , the *symptoms*,

(ii) derive all the absent and present base facts that correspond to the success of the symptoms, i. e. , the *causes*,

(iii) derive a transaction from the causes that restores the consistency of the database, i. e. , the *repair*.

Some additional papers of the research of Moerkotte are [Moe90], [MN91], [MR91] and [MS91].

7.1.3 Methods based on Adjustments of the Proof Procedure

Sadri and Kowalski In [KSS87] and [SK88] the SLDNF proof procedure is extended in order to simplify constraints in the proof procedure, instead of via a separate generation and selection phase. So, the bottom-up computation, such as in the case of the method of induced and potential updates, is incorporated into the proof procedure. While in SLDNF the top level goal can only be a denial, the proposed proof procedure allows any arbitrary rule, denial or negated fact. At some moment in this proof procedure the selected subgoal may be a potential deletion. In this case, before proceeding with the resolution process, first the effective induced deletions for this potential deletion are derived by the application of some meta-level inference rules. The full implementation of this proof procedure is given in [Sop86].

In [SI92] the proof method of Sadri and Kowalski is extended in order to cope with a broader class of deductive databases for which integrity constraints are checked.

Griefahn and Lüttringhaus In [GL90a] a contribution is made to handle existential quantifiers in constraints. Here, the deductive database is supposed to be normal. The completely goal-driven method presented in this paper can be seen as a variant of the method based on potential updates, although the potential updates are derived by a top-down computation instead of a bottom-up computation. By this computation all relevant potential updates are derived. Therefore, compared to the method based on potential updates, redundancy of the first kind is reduced to a minimum. In this method the structure of the deductive database is represented as an AND/OR tree, where a previous proof of consistency is stored by, what they call, a proper labelling of this tree. In this tree the instances of existentially quantified expressions in constraints used in the previous proof of the consistency of the database are stored. For instance, if $\exists X[p(X)]$ is some constraint that has to be fulfilled and an instance $p(a)$ satisfies this constraint in the database, then $p(a)$ is explicitly stored in that tree. As long as $p(a)$ remains in the database, this constraint does not have to be checked again. After each transaction the AND/OR tree is adjusted in order to reflect the new consistency proof. This adjustment proceeds top-down from the integrity constraints and is close to SLDNF-resolution. This method can be seen as integrity checking by SLDNF with the constraints as goal, where in the refutation information is used of a previous refutation and the information of the deductive database with respect to dependencies between predicates.

Celma, Casamayor, Decker et al. In [DC94] and [NDCC92] a proof procedure, which is related to SLDNF-resolution, is suggested for checking integrity constraints in deductive databases. Integrity constraints are represented as denials. However, the top-level goals consist of the (possibly rewritten) members of the transaction. In this Selection-driven Linear resolution procedure for Integrity Checking, called SLIC, forward reasoning is incorporated into its resolution based procedure. In SLIC a different kind of clauses is used, namely, extended clauses. Extended clauses are an extension of Horn clauses where a negated atom in the head of a clause is allowed as well. In SLIC negation in the body of extended rules is handled as negation by failure, while negation in the head of an extended rule is treated as classical negation. In a resolution step, besides a body literal also a head of an extended clause can be selected. The forward reasoning step, as in the generation and selection phase of the method of potential updates, is in SLIC established by resolving a head of an extended clause. This method is comparable to the method presented in [SK88] except that it improves the processing of deletions, which are handled by meta-level inference rules in [SK88]. In SLIC only induced deletions are computed for which the effectiveness is not tested. Therefore, the computation may be less accurate but can be done more efficiently as long as the number of ineffective updates remains relatively low. Here, the meta-level inference rules of [SK88] are incorporated into the resolution procedure of SLIC.

Asirelli et al. In [ABI89] the authors note that an inconsistency of the deductive database is introduced by an update. So, when proving that a deductive database is inconsistent, the refutation must contain a subgoal that is answered by the update. In order to accomplish such a proof, they altered the ordinary SLDNF-resolution into a SLDNF- resolution, where each proof must contain some specified clause, which they called the *driving clause*. Compared to the ordinary SLDNF-resolution, they used a new selection rule for selecting a new subgoal. When constraints

are represented as one overall denial, integrity checking is now established by resolving this denial as the top-level goal by this new resolution strategy, where the update is used as driving clause.

7.1.4 Methods based on Meta-logic Programming

Leuschel and Martens In [LM96] integrity checking is looked at from a different point of view, using a different, rather new concept of *partial deduction* (see [LM95a,LM95b,LS91]). The basis of partial deduction is a meta-programming approach in which the update is propagated through the intensional part of the deductive database. In the adjusted intensional database literals that are influenced by the update are replaced by their proper potential updates instantiating variables of the remainder of the rule that were instantiated by the potential update. So, instantiations by propagating potential updates are not lost, just as in the case of Asirelli et al. (see [AIM88]) which was described earlier in this chapter. This adjusted intensional database is used for evaluating the integrity constraints. This technique can be used also for *update patterns*. In that case, inconsistency rules are derived. Their results confirm that the analysis of redundancies and the usage and compilation of the inconsistency rules presented in this thesis were the proper way of handling integrity constraints efficiently. However, their results do not show how revised inconsistency rules can be derived by partial deduction. Until then integrity checking with partial deduction still contains redundancy of the third type. The results in this thesis show that the derivation of revised inconsistency rules by partial deduction must be encouraged. Further, recursion should be incorporated in this method in the way presented in this thesis.

7.1.5 Methods based on Adjustments of the Deductive Database

In this section two kinds of integrity checking methods that are characterized by an adjustment of the deductive database are distinguished, i. e. , the methods that are based on an adjustment of the set of rules and methods that are based on the adjustment of the set of constraints.

7.1.5.1 Methods based on Adjustments of Rules

Küchenhoff In [Küc91] a method for integrity constraint checking is incorporated into a method for computing the difference of two database states in a deductive database caused by a transaction. Deductive rules are adjusted in such a way that they can deduce this difference by using the query evaluator that is currently used. In [Küc91] the influence of an update of a predicate appearing in the body of an original deductive rule is made explicit and this bottom-up derivation is incorporated into the rule. Now, a set of rules is derived that is able to compute the difference of two database states caused by a transaction efficiently. He showed that integrity checking is related to this computation by reformulating constraints as rules of the form $inconsistent \leftarrow L_1, L_2, \ldots, L_n$. Now, those rules are adjusted just like the other deductive database rules, into a rule for computing the change in the predicate *inconsistent* efficiently. However, note that this predicate has no arguments; so, a change will not deliver new instances of this predicate. When the database is consistent before a transaction, the fact *inconsistent* cannot be derived. So, any difference in the predicate *inconsistent* implies that there exists an inconsistency.

The correspondence between this method and the method presented in this thesis is the incorporation of bottom-up information induced by general updates in base relations into the deductive database. However, the main difference between both methods is that in this method the bottom-up information is introduced in rules and in the method presented in this thesis this information is incorporated into the revised inconsistency rules. In [Bay92] it is noted that the method for efficiently deducing the difference between two consecutive database states may be helpful in other areas besides integrity checking. Note that in this integrity checking method no distinction between deductive rules and integrity constraints exists, while this distinction remains in the method based on revised inconsistency rules.

Olivé, Pastor, Urpí et al. In [Oli91], [Pas90] and [UO92] a method called the events method is presented. This method is closely related to the method in [Küc91]. In the events method rules are replaced by so called internal event rules in order to make derived updates explicit. Constraints are also represented as internal event rules. As a consequence, static and dynamic constraints can be handled in a uniform way. The consistency of the database is determined by making use of SLDNF-resolution. Denials of the inconsistency predicates, defined by the rules comprising the constraints, are used as top-level goals of the refutation. In [Oli91] Prolog rules were used for implementing this method.

In [Pas92] and [TO92] the events method is applied to the view update problem, where updates to views imply some adjustments to the base relation in order to comprise the update in these views; this also known as knowledge assimilation. This problem is recognized in relational databases as well as in deductive databases (see [Bry91,Dec92,GL90b,GL91,Hin95,KM90,LLS93,MKK+84, Ten95,Wüe93,Wüe92]. After assimilating the view update the database must be consistent again. In [Urp91] the events method applied to integrity checking is augmented by a new event, namely replacements.

7.1.5.2 Methods based on Adjustments of Constraints

Ling In [Lin87] integrity constraints and rules contain an extended form of negative literals, which we will call *normal form literals* (NF-literals), resulting in the class of the *normal form negative formulas* (NF-formulas). An NF-literal has the form $not(P)$ and is a normal form negative formula, where P is an atom or an expression of the form $A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge not(B_1) \ldots \wedge not(B_m)$, where A_i is an atom and $not(B_j)$ is a normal form negative formula. The interpretation of *not* is the same as in Prolog. Ling allows NF-formulas to appear in rules and constraints. Note that we get levels of NF-formulas. When the formulas are supposed to be safe, each variable in a NF-formula is instantiated by a variable in an atom appearing in the previous level of the NF-formula. This guarantees the full instantiation of a NF-formula at some level before evaluating it. When implementing normal form negative formulas the order of evaluation is expressed by the order of the subgoals in the NF-formula; how deeper the nesting of *not* the more the subgoal is placed to the right.

Integrity checking in relational databases is done in a similar way as in the method based on inconsistency rules, i. e. , updates directly instantiate the constraints that have to be evaluated. However, Ling generalizes his method to the deductive case by making integrity constraints relational by unfolding derived relations until they only contain base relations.

EXAMPLE 7.3 Consider the database of EXAMPLE 3.1. II_1 is replaced by the following iis:

$$\exists Y[father(X, Y), age(Y, N), N < 15, heavy\_job(Y)]$$

$$\exists Y[husband(Z, X), father(Z, Y), age(Y, N), N < 15, heavy\_job(Y)],$$

where the variables X, Z and N are instantiated by an update or by evaluating the indicator.

Note that when *age* was a derived relation defined in two separate rules, we had to unfold *age* as well, resulting in four inconsistency indicators. When integrity constraints consist of a lot of derived relations, which are defined by a lot of derived predicates, the unfolding of constraints may lead to an explosion of substitutes. Further, recursive definitions are forbidden in order to avoid infinite unfolding of the recursive definition. Note that in the method based on inconsistency rules only one derived predicate in the definition of an inconsistency indicator is unfolded, which makes the method more controlable. By the total unfolding of constraints, the method of Ling relies heavily on the Prolog evaluator, lacking the use of query optimization techniques. Further, as we have seen in 5.3.1, the method based on inconsistency rules allows the treatment of recursive relations.

Ling presents a solution to avoid redundant checking caused by replacements, by making separate integrity constraints in case of replacements. Because not all replacements are relevant to integrity constraints, a considerable reduction on the number of such constraints can be made. However, as we have seen, the inconsistency rules for insertions and deletions give enough clues to decide which inconsistency indicators must be re-evaluated when a fact is replaced.

Lee and Ling In the article of Lee and Ling, [LL94], relevant sets for each constraint are derived. Each set is used to determine for an arbitrary update whether the constraint must be checked or not. When it does not have to be checked, the gain is obvious, namely, the update can be accepted without any delay. Their method reduces the redundancy of the first kind. However, when the inconsistency indicator has to be checked, redundancies may still exist, depending on the method that is used.

Lee and Ling claim that their optimization technique can be added to any kind of integrity checking method. However, the redundancy of the first kind is already avoided in *FICCS*. One could say that their optimization technique for that kind of redundancy is integrated in the method based on revised inconsistency rules itself, because of the immediate triggering of inconsistency indicators by the update. The argument of the head of an inconsistency rule is strongly related to an element of the relevant set derived by Lee and Ling. However, they only prevent one particular part of redundancy of the first kind. For more details we refer to [LL94].

Wallace et al. In [LTW93], [Wal91] and [Wal92] besides the primitive updates, like insertions, deletions and replacements, some compound update types and their effect on integrity checking are studied. For instance, they deal with compound updates such as:

- *if* U_1 *then* U_2,
- *if* ⟨*condition*⟩ *then* U,
- *foreach* ⟨*vars* : *condition*⟩ *do* U(*vars*).

The first compound update states that an update U_2 is performed if update U_1 is performed. The second compound update states that an update U is performed when some condition is fulfilled. The third compound update states that for each instance of a variable fulfilling the given condition some update must be perfomed. For instance, suppose we want to insert tuples in a relation represented by a binary predicate p for each tuple appearing in the binary relation c, then this is expressed by the compound update *foreach* $c(X, Y)$ *do* $ins(p(X, Y))$, where we use the notation of updates of this thesis. Wallace presents in [Wal91, Wal92] a method which generates update procedures from the update and the set of constraints. In fact, from the update and the static constraints dynamic constraints are derived.

Seljée In [Sel94b], [Sel94c] and [Sel95b] the tests in [DW89b] were done for the method based on inconsistency rules. In this case the test results in [DW89b] and in [Sel94b], [Sel94c] and [Sel95b] were compared by making use of the test results for the naive method of checking integrity constraints, i. e. , a full check of constraints, implemented in both tests. The naive method is easy and straightforward to implement and will therefore not differ in both tests. Comparing the test results showed that in most cases the method based on inconsistency rules performs better than any of the methods tested in [DW89b].

In [Sel95a] these tests were done for the method based on revised inconsistency rules, which is an improvement of the method based on inconsistency rules. In this paper, the methods based on induced and potential updates were implemented as well, in order to get a real comparison in performance for several characteristic cases. The results of the tests show that in those cases in which the inconsistency rules did not perform as efficient as some other methods in [DW89b], the revised inconsistency rules will perform significantly better now. In fact, the method based on revised inconsistency rules will perform better than each method tested by Das and Williams in all tested cases. In this thesis, an elaborated analysis of the causes of most of the inefficiencies appearing in integrity checking methods is given. In [Sel93] and [Sel94a] tests are performed in a more complex example, where recursion in constraints is allowed as well.

The concept of integrity constraint is dropped in favour of the concept of inconsistency indicator, which is a different way of looking at the integrity of a database. Further, the Prolog proof procedure does not have to be adjusted in order to handle (revised) inconsistency rules. This method can easily be used in connection with a deductive database management system. In that way the method can use the query optimizer of the deductive database management system as well.

Other Related Research For some early contributions and for some minor contributions to the field of integrity constraint checking in deductive databases we refer to [AA89], [Deß93], [FG92], [GSUW94a], [GSUW94b], [Gup94], [Man90], [MMNR92], [Red93], [Rei88], [Rei90], [Wüe91] and [WY92].

7.2 Conclusions

In this section the main conclusions and advantages of the method based on revised inconsistency rules are given.

- We distinguished several classes of methods for checking integrity constraints. Each class of methods suffers from one or more redundancies in checking integrity constraints.
- We introduce a new method based on revised inconsistency rules that is optimal with respect to each of the redundancy types given in CHAPTER 3.
- Because of the incorporation of forward chaining information generated from arbitrary updates to the backward chaining revised inconsistency rules, the method based on revised inconsistency rules does not need a meta-interpreter. The revised inconsistency rules can be expressed naturally in any Prolog-like language of the deductive database management system.
- The revised inconsistency rules can be generated automatically at compile time. Therefore they can be optimized before any update of the database is made.
- The revised inconsistency rules can be adjusted incrementally; so, they do not have to be generated from scratch each time the set of rules or constraints is updated.
- The method based on revised inconsistency rules can be implemented in a straightforward manner, as CHAPTER 6 and the appendix of this thesis show.

References

[AA89] N. AZARMI AND M. AZMOODEH. Logic Databases and Integrity Constraints. In KEVIN P. JONES, editor, *Prospects for Intelligent Retrieval, Informatics 10, Proceedings of a conference jointly sponsored by Aslib, the Aslib Informatics Group and the Information Retrieval Specialist Group of the British Computer Society*, pages 295–302, Cambridge, March 1989. King's College.

[ABI89] P. ASIRELLI, C. BILLI AND P. INVERARDI. Selective Refutation of Integrity Constraints in Deductive Databases. In J. DEMETROVICS AND B. THALHEIM, editors, *Lecture Notes in Computer Science*, volume 364, pages 1–11, Visegrád, Hungary, June 1989.

[AdSM85] PATRICIA ASIRELLI, MICHÈLE DE SANTIS AND MAURIZIO MARTELLI. Integrity Constraints in Logic Databases. *Journal of Logic programming*, 3:221–232, 1985.

[AIM88] P. ASIRELLI, P. INVERARDI AND A. MUSTARO. Improving Integrity Constraint Checking in Deductive Databases. In Gyssens et al. [GPvG88], pages 72–86.

[Bay92] PETRA BAYER. Update Propagation for Integrity Checking, Materialized View Maintenance and Production Rule Triggering. Technical Report 92-10, ECRC GMBH, Arabellastr. 17 D-8000 München 81, Germany, 1992.

[BD88] FRANÇOIS BRY AND HENDRIK DECKER. Préserver l'Integrité d'une Base de Données Déductive: une Méthode et son Implémentation. In *Proceedings of the 4èmes Journées Bases de Données Avancées (BDA)*, May 1988.

[BDM87] FRANÇOIS BRY, HENDRIK DECKER AND RAINER MANTHEY. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In J. W. SCHMIDT, S. CERI AND M. MISSIKOFF, editors, *Advances in Databases Technology, EDBT '88; Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 488–505, Venice, Italy, November 1987. also: ECRC Technical Report KB-16.

[Bla90] A. BLASER, editor. *Database Systems of the90s; International Symposium, Proceedings*, volume 466 of *Lecture Notes in Computer Science*, Mügelsee, Berlin, FRG, November 1990.

[BM86] FRANÇOIS BRY AND RAINER MANTHEY. Checking Consistency of Database Constraints: a Logical Basis. In WESLEY CHU, GEORGE GARDARIN AND SETSUO OHSUGA, editors, *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 13–20, Kyoto, Japan, 1986.

[BMM91] FRANÇOIS BRY, RAINER MANTHEY AND BERN MARTENS. Integrity Verification in Knowledge Bases. In *Proceedings of the Second Russian Conference on Logic Programming*, pages 114–139, Saint Petersburg, Russia, September 1991.

[Bry87] F. BRY. Maintaining Integrity of Deductive Databases. Technical Report KB-45, ECRC, München, July 1987.

[Bry91] FRANÇOIS BRY. Intensional Updates: Abduction via Deduction. In Delobel et al. [DKM91], pages 561–578.

[BS89] H. A. BLAIR AND V. S. SUBRAHMANIAN. Paraconsistent Logic Programming. *Theoretical Computer Science*, 68(2):135–154, 1989.

[CCD93] MATILDE CELMA, JUAN CARLOS CASAMAYOR AND HENDRIK DECKER. Improving Integrity Checking by compiling Derivation paths. In Orlowska and Papazoglou [OP93], pages 145–160.

[CCM+91] M. CELMA, J. C. CASAMAYOR, L. MOTA, M. A. PASTOR AND F. MARQUÉS. A Derivation Path Recording Method for Integrity Checking in Deductive Databases. In *Advances in Database Research; Proceedings of the Second International Workshop on the Deductive Approach to Information Systems and Databases*, pages 185–203, Aiguablava-Costa Brava, Catalonia, September 1991.

[CGMD94] M. CELMA, C. GARCÍA, L. MOTA AND H. DECKER. Comparing and Synthesizing Integrity Checking Methods for Deductive Databases. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 214–222, Houston, Texas, February 1994. IEEE Computer Society Press.

[CM92] M. CELMA AND L. MOTA. Foundations of Simplified Integrity Checking Reviewed. In A. M. TJOA AND I. RAMOS, editors, *Proceedings of the 3rd International Conference, DEXA'92; Database and Expert Systems Applications*, pages 90–95, Valencia, Spain, 1992. Springer-Verlag.

[Das90] S. K. DAS. *Integrity Constraints in Deductive Databases*. PhD thesis, Dissertation at the Computer Science Department, Heriot-Watt University, Edinburgh, 1990.

[Das91] S. K. DAS. Specifying Deductive Databases and Integrity Constraints in Metalogic. In *Specifications of Database Systems*, pages 333–340, 1991.

[Das92] S. K. DAS. *Deductive Databases and Logic programming*. Addison-Wesley, 1992.

[DC94] HENDRIK DECKER AND MATILDE CELMA. A Slick Procedure for Integrity Checking in Deductive Databases. In *Proceedings of the Eleventh Int'l Conference on Logic Programming; ICLP'94*, volume 31, pages 456–469, 1994.

[Dec87] H. DECKER. Integrity Enforcement on Deductive Databases. In LARRY KERSCHBERG, editor, *Expert Database Systems: Proceedings from the First International Conference*, pages 271–285. Charleston, Sc., 1987.

[Dec92] HENDRIK DECKER. Knowledge Assimilation in Deductive Databases. In *Proceedings of the Third International Workshop on the Deductive Approach to Information Systems and Databases*, pages 217–247, Roses-Costa Brava, Catalonia, September 1992.

[Deß93] STEFAN DESSLOCH. *Semantic Integrity in Advanced Database Management Systems*. PhD thesis, Dissertation at the University of Informatics of Kaiserslautern, 1993.

[DGK+95] HENDRIK DECKER, ULICH GESKE, ANTONIS C. KAKAS, CHIAKI SAKAMA, DIETMAR SEIPEL AND TONI URP, editors. *Deductive Databases and Logic Programming, Abduction in Deductive Databases and Knowledge-Based Systems*, Shonan Village Center, Japan, June 1995.

[DKM91] C. DELOBEL, M. KIFER AND Y. MASUNAGA, editors. *Deductive and Object-Oriented Databases; Proceedings of the Second International Conference, DOOD'91*, volume 566 of *Lecture Notes in Computer Science*, Munich, Germany, December 1991.

[DW89a] S. K. DAS AND M. H. WILLIAMS. Integrity Checking Methods in Deductive Databases: A Comparative Evaluation. In M. H. WILLIAMS, editor, *Proceedings of the Seventh British National Conference on Databases*, pages 85–116. Cambridge University Press, 1989.

[DW89b] S. K. DAS AND M. HOWARD WILLIAMS. A Path Finding Method for Constraint Checking in Deductive Databases. *Data & Knowledge Engineering*, 4:223–244, 1989.

[FG92] M. M. FONKAM AND W. A. GRAY. Employing Integrity Constraints for Query Modification and Intensional Answer Generation in Multi-database Systems. In P. M. D. GRAY AND R. J. LUCAS, editors, *Lecture Notes in Computer Science*, volume 618 of *Lecture Notes in Computer Science*, pages 244–260, Aberdeen, Scotland, July 1992.

[GL90a] ULRIKE GRIEFAHN AND STEFAN LÜTTRINGHAUS. Top-Down Integrity Constraint Checking for Deductive Databases. In DAVID H. D. WARREN AND PETER SZEREDI, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 130–144, Jerusalem, 1990.

[GL90b] A. GUESSUOM AND J. W. LLOYD. Updating Knowledge Bases. *New Generation Computing*, 8(1):71–89, 1990.

[GL91] A. GUESSUOM AND J. W. LLOYD. Updating Knowledge Bases II. *New Generation Computing*, 10:73–100, 1991.

[GPvG88] M. GYSSENS, J. PAREDAENS AND D. VAN GUCHT, editors. *Proceedings of the Second International Conference on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, Bruges, Belgium, August-September 1988.

[GS95] JOHN GRANT AND V. S. SUBRAHMANIAN. Reasoning in Inconsistent Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):177–189, February 1995.

[GSUW94a] ASHISH GUPTA, YEHOSHUA SAGIV, JEFFREY D. ULLMAN AND JENNIFER WIDOM. Constraint Checking with Partial Information. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 44–55, Minneapolis, MN., May 1994. ACM Press, New York.

[GSUW94b] ASHISH GUPTA, YEHOSHUA SAGIV, JEFFREY D. ULLMAN AND JENNIFER WIDOM. Efficient and Complete Tests for Database Integrity Constraint Checking. In ALAN BORNING, editor, *Second International Workshop, Principles and Practice of Constraint Programming, PPCP'94*, volume 874 of *Lecture Notes in Computer Science*, pages 173–180, Rosario, Orcas Island, WA, USA, May 1994. Springer-Verlag.

[Gup94] ASHISH GUPTA. *Partial Information based Integrity Checking*. PhD thesis, Stanford University, Stanford, December 1994.

[Hin95] KNUT HINKELMANN. Knowledge-Base Rewriting for Bottom-Up Abduction and Integrity Checking. In Decker et al. [DGK+95], pages 127–141.

[Hum92] RENATE HUMS. *Effiziente Integritätssicherung in Deduktiven Datenbanksystemen durch Logikprogrammtransformation*. PhD thesis, Institut für Informatik der Technischen Universität München, München, December 1992.

[JJ91] MANFRED JEUSFELD AND MATTHIAS JARKE. From Relational to Object-Oriented Integrity Simplification. In Delobel et al. [DKM91], pages 460–477.

[JK90] MANFRED JEUSFELD AND EVA KRÜGER. Deductive Integrity Maintenance in an Object-Oriented Setting. MIP 9013, Technische Berichte der Fakultät für Mathematik und Informatik Universität Passau, 1990.

[KM90] A. C. KAKAS AND P. MANCARELLA. Database Updates Through Abduction. In DENNIS MCLEOD, RON SACKS-DAVIS AND HANS SCHEK, editors, *Proceedings of the Sixteenth Conference on Very Large Data Bases*, pages 650–661, Brisbane, Australia, 1990.

[KSS87] ROBERT KOWALSKI, FARIBA SADRI AND PAUL SOPER. Integrity Checking in Deductive Databases. In PETER M. STOCKER AND WILLIAM KENT, editors, *Proceedings of the Thirteenth Conference on Very Large Data Bases*, pages 61–69, Brighton, England, August 1987.

[Küc91] V. KÜCHENHOFF. On the Eficient Computation of the Difference Between Consecutive Database States. In Delobel et al. [DKM91], pages 478–502.

[Lin87] TOK-WANG LING. Integrity Constraint Checking in Deductive Databases using the Prolog not-predicate. *Data & Knowledge Engineering*, 2:145–168, 1987.

[LL94] SIN YEUNG LEE AND TOK WANG LING. Improving Integrity Constraint Checking for Stratified Deductive Databases. In DIMITRIS KARAGIANNIS, editor, *Lecture Notes in Computer Science*, volume 856, pages 591–600, Athens, Greece, September 1994. Springer-Verlag.

[LLS93] DOMINIQUE LAURENT, V. PHAN LUONG AND NICOLAS SPYRATOS. Updating Intensional Predicates in Deductive Databases. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 14–21, Vienna, Austria, April 1993.

[LM95a] M. LEUSCHEL AND B. MARTENS. Partial Deduction of the Ground Representation and its Application to Integrity Checking. Technical Report CW210, Departement Computerwetenschappen, K.U.Leuven, Belgium, April 1995. Updated August 1995.

[LM95b] M. LEUSCHEL AND B. MARTENS. Partial Deduction of the Ground Representation and its Application to Integrity Checking. In JOHN LLOYD, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, Portland, USA, December 1995. MIT Press.

[LM96] ALON Y. LEVY AND INDERPAL SINGH MUMICK. Reasoning with Aggregation Constraints. In P. APERS, M. BOUZEGHOUB AND G. GARDARIN, editors, *Advances in Databases Technology, EDBT '96; Proceedings of the Fifth International Conference on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 514–534, Avignon, France, March 1996.

[LS91] J. W. LLOYD AND J. C. SHEPHERDSON. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[LSC91] GUY M. LOHMAN, ALMICAR SERNADAS AND RAFAEL CAMPS, editors. *Proceedings of the Seventeenth Conference on Very Large Data Bases*, Barcelona, Spain, 1991.

[LST87] J. W. LLOYD, E. A. SONENBERG AND R. W. TOPOR. Integrity Constraint Checking in Stratified Databases. *Journal of Logic Programming*, 4:331–343, 1987.

[LT85] J. W. LLOYD AND R. W. TOPOR. A Basis for Deductive Database Systems. *Journal of Logic Programming*, 2:93–109, 1985.

[LT86] J. W. LLOYD AND R. W. TOPOR. A Basis for Deductive Database Systems II. *Journal of Logic Programming*, 3(1):55–67, 1986.

[LTW93] MICHAEL LAWLEY, RODNEY TOPOR AND MARK WALLACE. Using Weakest Preconditions to Simplify Integrity Constraint Checking. In Orlowska and Papazoglou [OP93], pages 161–170.

[Man90] RAINER MANTHEY. Integrity and Recursion: Two Key Issues for Deductive Databases. In D. KARAGIANNIS, editor, *Proceedings of the First Workshop on Information Systems and Artificial Intelligence: Integration Aspects*, volume 474 of *Lecture Notes in Computer Science*, pages 104–126, Ulm, FRG, March 1990.

[MB86] BERN MARTENS AND MAURICE BRUYNOOGHE. Integrity Constraint Checking in Deductive Databases. In LARRY KERSCHBERG, editor, *Expert Database Systems: Proceedings from the First International Workshop*, pages 567–601. Charleston, Sc., 1986.

[MK88] GUIDO MOERKOTTE AND S. KARL. Efficient Consistency Checking in Deductive Databases. In Gyssens et al. [GPvG88], pages 118–128.

[MKK+84] TAIZO MIYACHI, SUSUMU KUNIFUJI, HAJIME KITAKAMI, KOICHI FURUKAWA, AKIKAZU TAKEUCHI AND HARUO YOKOTA. A Knowledge Assimilation Method for Logic Databases. *New Generation Computing*, 2(4):385–404, 1984.

[ML91] GUIDO MOERKOTTE AND PETER C. LOCKEMANN. Reactive Consistency Control in Deductive Databases. *ACM Transactions on Database Systems*, 16(4):670–720, December 1991.

[MMNR92] SALVADOR VILENA MORALES, EMILIA RUIZ MARTÍN, CECILIA DELGADO NEGRETE AND BUENAVENTURA CLARES RODRIGUEZ. Efficient Implementation of Deductive Database Based on the Method of Potential Effects. In *Proceedings of the 3rd International Conference, DEXA'92; Database and Expert Systems Applications*, pages 545–546, 1992.

[MN91] GUIDO MOERKOTTE AND A. NEUFELD. Generating Consistent Test Data for a Variable Set of General Consistency Constraints. In Lohman et al. [LSC91].

[Moe90] GUIDO MOERKOTTE. Inkonsistenzen in Deduktiven Datenbanken, Diagnose und Reparatur. Informatik Fachbericht 248, Universität Karlsruhe, 1990.

[MR91] GUIDO MOERKOTTE AND KARL RÖSCH. On the compilation of Consistency Constraints (Extended Abstract). In *Proceedings of the Second International Workshop on the Deductive Approach to Information Systems and Databases*, pages 174–184, Aiguablava-Costa Brava, Catalonia, September 1991.

[MS91] GUIDO MOERKOTTE AND P. SCHMITT. Analysis and Repair of Inconsistencies in Deductive Databases. *Journal of Logic Programming*, 1991.

[MSW91] T. MURATA, V. S. SUBRAHMANIAN AND T. WAKAYAMA. A Petri Net Model for Reasoning in the Presence of Inconsistency. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):281–292, September 1991.

[NDCC92] GEORG NÜSSEL, HENDRIK DECKER, MATILDE CELMA AND JUAN CARLOS CASAMAYOR. A Complete Proof Procedure for efficient Integrity Checking in Deductive Database Systems. In *Proceedings of the Third International Workshop on the Deductive Approach to Information Systems and Databases*, pages 199–216, Roses-Costa Brava, Catalonia, September 1992.

[Nic79] J. M. NICOLAS. A Property of Logical Formulas corresponding to Integrity Constraints on Data Base Relations. In *Proceedings of the Workshop on Formal Bases for Data Bases*, Toulouse, 1979.

[Nic82] J. M. NICOLAS. Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, 18(3):227–253, 1982.

[NY78] J. M. NICOLAS AND K. YAZDANIAN. Integrity Checking in Deductive Data Bases. In H. GALLAIRE AND J. MINKER, editors, *Logic and Databases*, pages 325–346, New York, 1978. Plenum Press NY.

[Oli91] ANTONI OLIVÉ. Integrity Constraint Checking in Deductive Databases. In Lohman et al. [LSC91], pages 513–523.

[OP93] M. E. ORLOWSKA AND M. PAPAZOGLOU, editors. *Advances in Database Research; Proceedings of the Fourth Australian Database Conference*, Brisbane, Australia, February 1993.

[Pas90] JOAN A. PASTOR. The Internal Events Method for Integrity Constraint Enforcement in Deductive Databases. In *Proceedings of the Second International Workshop on the Deductive Approach to Information Systems and Databases*, pages 111–140, Agaro, Spain, 1990.

[Pas92] JOAN A. PASTOR. Deriving Consistency-Preserving Transaction Specifications for (View-)Updates in Relational Databases. In *Proceedings of the Third International Workshop on the Deductive Approach to Information Systems and Databases*, pages 275–300, Roses-Costa Brava, Catalonia, September 1992.

[Red93] SWARUP REDDI. Integrity Constraint Enforcement in the Functional Database Language PFL. In M. WORBOYS AND A. F. GRUNDY, editors, *Proceedings of the Eleventh British National Conference on Databases*, Lecture Notes in Computer Science, pages 238–257, Keele, UK, July 1993. Springer-Verlag.

[Rei88] RAYMOND REITER. On Integrity Constraints. In *Proceedings of Theoretical Aspects of Reasoning About Knowledge*, pages 97–112, 1988.

[Rei90] RAYMOND REITER. Integrity Constraints for Knowledge Bases. In R. A. MEERSMAN, ZHONGZHI SHI AND CHEN-HO KUNG, editors, *AI in Databases and Information Systems (DS-3)*, pages 3–12. IFIP, Working Conference on the Role of Artificial Intelligence in Database and Information Systems, 1990.

[Sel93] RON R. SELJÉE. Expert Database Systems with Integrity Constraint Checking. Technical report, pages 1–28. Centre for Knowledge Engineering (CIBIT), Utrecht, The Netherlands, 1993.

[Sel94a] RON R. SELJÉE. Expert Database Systems with Integrity Constraint Checking; APPENDIX. Technical report, pages 29–55. Centre for Knowledge Engineering (CIBIT), Utrecht, The Netherlands, 1994.

[Sel94b] RON R. SELJÉE. Integrity Constraint Checking for Updates in Deductive Databases; A Different Approach. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 52:23p, 1994.

[Sel94c] RON R. SELJÉE. Integrity Constraint Checking in Deductive Databases. Computing Science Reports 94-13:1–34, Department of Computing Science, TU Eindhoven, 1994.

[Sel95a] RON R. SELJÉE. Deductive Databases and Integrity Constraint Checking. Computing Science Reports 95-11:1–36, Department of Computing Science, TU Eindhoven, 1995.

[Sel95b] RON R. SELJÉE. A New Method for Integrity Constraint Checking in Deductive Databases. *Data & Knowledge Engineering*, 15(1):63–102, March 1995.

[SI92] KEN SATOH AND NOBORU IWAYAMA. A Correct Goal-directed Proof Procedure for a General Logic Program with Integrity Constraints. In ALAN BORNING, editor, *Proceedings of the Third International Workshop of Extensions of Logic Programming, ELP'92*, volume 660 of *Lecture Notes in Computer Science*, pages 24–44, Bologna, Italy, February 1992. Springer-Verlag.

[SK88] FARIBA SADRI AND ROBERT KOWALSKI. A Theorem-Proving Approach to Database Integrity. In J. MINKER, editor, *Foundations of Deductive Databases and Logic Programming*, pages 313–362, Los Altos, 1988. Morgan Kaufmann.

[Sop86] P. J. R. SOPER. *Integrity Checking in Deductive Databases*. PhD thesis, Department of Computing, Imperial College, University of London, 1986.

[Ten95] ERNEST TENIENTE. An Abductive Framework to Handle Consistency-preserving Updates in Deductive Databases. In Decker et al. [DGK+95], pages 111–125.

[TO92] ERNEST TENIENTE AND ANTONI OLIVÉ. The Events Method for View Updating in Deductive Databases. In A. PIROTTE, C. DELOBEL AND G. GOTTLOB, editors, *Advances in Data Technology-EDBT '92; Proceedings of the Third International Conference on Extending Database Technology*, number 580 in Lecture Notes in Computer Science, page 16. Springer-Verlag, 1992.

[Ull85] J. D. ULLMAN. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1985.

[UO92] TONI URPÍ AND ANTONI OLIVÉ. A Method for Change Computation in Deductive Databases. In LI-YAN YUAN, editor, *Proceedings of the Eighteenth International Conference on Very Large Databases*, pages 225–237, Vancouver, Canada, 1992.

[Urp91] TONI URPÍ. An Approach to Monitoring Changes in Deductive Databases. In *Proceedings of the Second International Workshop on the Deductive Approach to Information Systems and Databases*, pages 87–113, Aiguablava-Costa Brava, Catalonia, June 1991.

[Wal91] MARK WALLACE. Compiling Integrity Checking into Update Procedures. In *Proceedings of the Thirteenth IJCAI*, pages 903–908, Sydney, Australia, August 1991.

[Wal92] MARK WALLACE. Compiling Integrity Checking into Update Procedures. Technical Report 92-19, ECRC GMBH, Arabellastr. 17 D-8000 München 81, Germany, 1992.

[Wüe91] BEAT WÜETHRICH. *Large Deductive Databases with Constraints*. PhD thesis, Swiss Federal Institute of Technology Zürich, 1991.

[Wüe92] BEAT WÜETHRICH. Update Realizations Drawn from Knowledge Base Schemas and Executed by Dialog. Technical Report 92-4, ECRC GMBH, Arabellastr. 17 D-8000 München 81, Germany, 1992.

[Wüe93] BEAT WÜETHRICH. On Updates and Inconsistency Repairing in Knowledge Bases. In *Proceedings of the Nineth IEEE International Conference on Data Engineering*, pages 608–616, Vienna, Austria, April 1993.

[WY92] KE WANG AND LI YAN YUAN. Preservation of Integrity Constraints in Definite DATALOG Programs. *Information Processing Letters*, 44(4):185–193, December 1992.

Appendix A

DHIS; A Case Study using *FICCS*

In the appendix of this thesis, a case of a deductive database with integrity constraint checking by *FICCS* is given, namely, an experimental Deductive database concerning a Hospital Information System, called DHIS. This case is implemented and tested by using revised inconsistency rules. For different sets of rules and inconsistency indicators the sets of related revised inconsistency rules are available and do not have to be generated during the integrity check. Besides the method based on revised inconsistency rules, two other methods for checking the integrity of the deductive database are used, namely, the method based on induced updates and the method based on potential updates. The implementations of these methods are also given in this appendix.

The implementation is a simplification of the implementation presented in CHAPTER 6. For instance, the transactions in question do not contain rule updates or inconsistency indicator updates. Further, the database is constructed in such a way that query evaluation is only necessary in the updated database and not in the current database. The implementation used in the appendix is not described in detail here, because its main principles are comparable to those described previously in this thesis.

A.1 Implementations of Integrity Constraint Checking Methods

All presented integrity checking methods are initialized by a transaction. For each update in the transaction, called T, the integrity checking method tries to find a proof for inconsistency. In this implementation an update is represented by a list consisting of two members; the first member is a sign, + in case of an insertion and − in case of a deletion. The inconsistency checks in all methods are controlled by the following logic program.

```
new(Method,T) :-
  proof_inconsistent(Method,T),
  !.
new(Method,T) :-
  consistent(Method,T),
```

```
  !.

proof_inconsistent(Method,T) :-
  (inconsistent(Method,T) ;
   proof_inconsistent(Method,T)).

consistent(Method,T) :-
  write('Transaction '),
  write(T),
  write(' is accepted.'),
  nl,
  write('The method that was used is '),
  write(Method),
  nl,
  commit(T).
```

Each update in the transaction may lead to an inconsistent state of the database. If such an update does not exist, then `consistent(T)` will succeed and the transaction is accepted. In that case, the predicate `commit` handles the execution of the transaction. On the other hand, if such an update exists, then the predicate inconsistent will succeed and no committing action is taken. Now, the only action taken is the generation of a simple report that states which facts are causing the inconsistency, which inconsistency rule succeeds, etc..

A.1.1 Implementation of the Method based on Induced Updates

```
% The method based on induced updates.

inconsistent(induced,T) :-
  member_list(all_exist,list(T),el(Update)),
  induced(Ind,Update,T),
  ii(N,II),
  match_ind(II,II1,Ind,+),
  new(II1,T),
  report([N,II,Update,Ind],T,induced).

induced(X,X,T).
induced(Ind,X,T) :-
  forward_ind(Ind1,X,T),
  \+ Ind1 = X,
  induced(Ind,Ind1,T).

forward_ind([S,H],Y,T) :-
  rule_forward(N,head(H),body(B)),
```

```
  match_ind(B,B1,Y,S),
  new(B1,T).

match_ind([X|L],L,Y,S) :-
  match(X,Y,S).
match_ind([X|L],[X|L1],Y,S) :-
  match_ind(L,L1,Y,S).
```

A.1.2 Implementation of the Method based on Potential Updates

```
% The method based on potential updates.

inconsistent(potential,T) :-
  member_list(all_exist,list(T),el(Update)),
  Update =.. [F|VarL],
  potential(Pot,Update,VarL),
  ii(N,II),
  match_pot(II,II1,Pot,+),
  new(II,T),
  report([N,II,Update,Pot],T,potential).

potential(X,X,VarL).
potential(Pot,[SX,X],VarL) :-
  forward_pot([SP1,Pot1],[SX,X]),
  \+ (SP1 = SX,unifiable_Lit(Pot1,X,VarL)),
  potential(Pot,[SP1,Pot1],VarL).

forward_pot([S,H],Y) :-
  rule_forward(N,head(H),body(B)),
  match_pot(B,B1,Y,S).

match_pot([X|L],L,Y,S) :-
  match(X,Y,S).
match_pot([\+ var(Z)|L],L1,Y,S) :-
  match_pot(L,L1,Y,S).
match_pot([X|L],[X|L1],Y,S) :-
  \+ X = (\+ var(Z)),
  match_pot(L,L1,Y,S).
```

A.1.3 Implementation of the Method based on Inconsistency Rules

In the method based on inconsistency rules we add those rules to the database. If the rules and the inconsistency rules were represented as Prolog rules, then the Prolog evaluator could perform the

consistency check. However, the previously mentioned query-evaluator is used. Therefore, the database rules are represented as Prolog facts, see the database rules as presented in CHAPTER 6. Further, a minor adjustment of the inconsistency rules, represented as Prolog rules, is made. Instead of inconsistency rules of the form:

```
inc_rule(⟨update⟩) :-
    ⟨goal1⟩,⟨goal2⟩, ... ,⟨goaln⟩.
```

inconsistency rules are now expressed by:

```
inc_rule(⟨update⟩,T) :-
    new([⟨goal1⟩,⟨goal2⟩, ... ,⟨goaln⟩],T).
```

Because the database rules now have to be represented as general Prolog facts, the forward reasoning in the methods based on induced and potential updates is easier. The inconsistency check in each method is initiated by the `inconsistent` predicate. Now the whole check is performed by using the rule:

```
% The method based on inconsistency rules.

inconsistent(incrules,T) :-
  member_list(all_exist,list(T),el(Update)),
  inc_rule(Update,T),
  report(Update,T,incrules),
  !.
```

The check is performed by backtracking on every success of `member_list` as long as the evaluation of `inc_rule` for some update fails.

A.1.4 Comparing the Implementations

The logic programs implementing the method based on induced updates and the method based on potential updates are presented in A.1.1 resp. A.1.2. When comparing both implementations, we see that besides using different predicates, `induced` and `potential`, the methods are alike. Note, the difference between the code:

```
inconsistent(induced,T) :-
  member_list(all_exist,list(T),el(Update)),
  induced(Ind,Update,T),
  ii(N,II),
  match_ind(II,II1,Ind,+),
  new(II1,T),
  report([N,II,Update,Ind],T,induced).
```

and the code:

```
inconsistent(potential,T) :-
  member_list(all_exist,list(T),el(Update)),
  Update =.. [F|VarL],
  potential(Pot,Update,VarL),
  ii(N,II),
  match_pot(II,II1,Pot,+),
  new(II,T),
  report([N,II,Update,Pot],T,potential).
```

In fact, the only difference is de first variable in `new`. Let `II` represent an inconsistency indicator. After matching the update `Ind`, representing an induced update, resp. `Pot`, representing the potential update, with a literal in `II`, `II1` contains the instantiated inconsistency indicator without the matched update. As stated in CHAPTER 2, induced updates matching a literal in the instantiated indicator do not have to be evaluated any more and can be left out from the expression, while potential updates must be evaluated and cannot be left out from the expression. So, in the induced case it is sufficient to query `II1` instead of `II` in the potential case.

Note also the difference between the predicates `forward_ind` and `forward_pot`, which are intended to reason forward from an induced resp. potential update `Y` in order to find all other induced resp. potential updates. Here, `[S,H]` represents the derived (induced resp. potential) update.

```
forward_ind([S,H],Y,T) :-
  rule_forward(N,head(H),body(B)),
  match_ind(B,B1,Y,S),
  new(B1,T).
```

and

```
forward_pot([S,H],Y) :-
  rule_forward(N,head(H),body(B)),
  match_pot(B,B1,Y,S).
```

In both methods, after the evaluation `match`, `B1` represents a potential update with respect to `Y`. Note that `B1` is evaluated in the updated database in the first case, while it is not evaluated in the second case. If in the first method the evaluation in the updated database succeeds, then `B1` becomes an induced update. However, when the query fails and no other induced update from `Y` can be found by backtracking, the search for induced updates from `Y` stops here. In the second method, `B1` is not evaluated and all potential updates from `Y` are found. Note that `forward_pot` does not have to contain the transaction as one of its arguments like in `forward_ind`, because the evaluation of the potential update in the updated database, for which the transaction is used, is skipped or postponed to the evaluation of a relevant inconsistency indicator.

A.2 DHIS; A Deductive Hospital Information System extended with $\mathcal{FICCS}$

In this section, the implementation of an example Deductive Hospital Information System (DHIS) is presented, which is based on the hospital database described by *E.O. de Brock* in his book about semantical databases (see [Bro78]). Knowledge of Hospital Information Systems, a hospital's organisational structure, medical information, etc., is well-documented. The hospital environment is chosen for two reasons, namely:

(i) the domain is semantically very rich; therefore, a lot of rules and integrity constraints can be stated for the data,

(ii) it shows the relevance of automated integrity checking.

In hospitals a lot of decisions are made based on a lot of constraints. For instance, the condition of a patient, possible available treatments, illness of a patient, number of beds in a hospital, patients and their allergy for medicines, etc.. Decisions may turn out to be wrong or cannot be made immediately or even not at all because they do not take all constraints into account. There are several reasons for neglecting one or more constraints, namely:

(i) the constraints are spread all over the hospital,

➢ for instance, the test results of the lab, specialists who must diagnose a patient, the available time they have and the available medication are important for making a decision for a proper treatment,

(ii) the checking of the constraints is time-consuming and therefore is not evaluated,

➢ for instance, analysing the treatment of a relative of a patient, who has had the same illness, and adjust the treatment using these facts, or when the treatment has to be done acutely and no time has to be wasted for checking some constraints,

(iii) the constraints are only known to experts, who may not be available at that time,

➢ for instance, when unexpectedly a patient gets an attack at night and only a nurse is available,

(iv) a constraint is evaluated wrongly or not at all by a human because of a lot of pressure,

➢ for instance, giving the patient a wrong medicine or too much of some medicine in acute cases,

A Hospital Information System supporting those decisions by making constraints accessible and automatically checking those constraints efficiently is needed. This appendix tries to show that a deductive database with integrity constraint checking can be helpful in supporting those decisions.

This system does not pretend to be a working database system that can be used in a real hospital environment right away. In order to accomplish this a lot of knowledge acquisition on experts working in hospitals has to be done in order to get the constraints on the data clear. Here, the

system is built in order to give an idea of its working, its relevance and its performance. This appendix contains the highlights of the appendix of [Sel93], which appeared as a separate report (see [Sel94]).

A.2.1 Data model of DHIS

In the first part of this section the data-structures, which are used in the database, are given. Some of these data-structures make use of other data-structures. The NAR-structure is a structure which gives the name, adress and residence of a person, department, etc.. The DATE-structure represents the date. BOOL represents yes and no. The DATE- and BOOL-structures are structures which are taken separately because in databases they have a separate data type which is resp. DATE and BOOL. The TEL structure is defined to represent the telephone number. The structures NAR and TEL are defined separately from the other structures of the database in order to avoid their repetitive definition in these structures. In the second part, the set of rules and inconsistency indicators of the database is given.

A.2.2 Datastructures

This section gives a short overview of the datastructures used in DHIS. It is intended to be self-explanatory for someone familiar with hospital information systems. However, non-experts in the field should be able to understand the global idea of how the database is supposed to look like. The NAR and TEL structures are as follows:

STRUCTURE NAR

first_name:	char(20)
last_name:	char(20)
street_name:	char(15)
street_number:	number(3)
p. o. _box:	char(7)
residence:	char(25)

STRUCTURE TEL

first_dial:	char(6)
second_dial:	number(8)

The other structures can be found below:

STRUCTURE patients

patient_id:	number(6)
name/adress/residence:	NAR
date_of_birth:	DATE
date_of_registration:	DATE
blood_type:	$[O, A, B, AB]$

Rhesus_factor:	$[+, -]$
sex:	$[M, V]$
family_doctor_id:	[1..99]

STRUCTURE employees

emp_id:	number(4)
name/adress/residence:	NAR
department_nr:	[1..100]
internal_telephone:	number(4)
employee_code:	$[SP, MP, NMP]$

STRUCTURE specialists

sp_id:	number(3)
locum_tenens_id_nr:	number(3)
specialisation:	char(40)
telephone:	TEL
fee:	[8000..)
hours/week:	[0..50]
beds:	number
in_service:	BOOL

STRUCTURE staff members

st_id:	number(4)
head_nr:	number(4)
half_days/week:	[1..10]
salary:	[0..*maxsal*]
days_of_absence:	[0..290]

STRUCTURE medical staff members

ms_id:	number(4)
telephone:	TEL
schooling_code:	char(2)

STRUCTURE not medical staff members

nms_id:	number(4)
function_code:	char(10)
number_days_off:	[15..30]
payment:	number

STRUCTURE family doctor

family_doctor_id:	[1..99]
name/adress:	NAR
sex:	$[M, V]$

telephonenumber: TEL
number_of_patients: number(3)

STRUCTURE family relation
pat_id: number(6)
relation: [*child*, *married*]
relative_id: number(6)

STRUCTURE room
room_id: number(4)
department_nr: [1..100]
area: number
status: [*AV*, *PL*, *NA*]
room_type: [*MED*, *ADM*, *LAB*, *SER*, *OTH*]
number_of_beds: [0..15]

STRUCTURE admission
patient_id: number(6)
day_of_admission: DATE
room_id: number(4)
specialist_id: number(3)
discharged: BOOL
deceased: BOOL

STRUCTURE treatment
treatment_code: number(4)
treatment_name: char(50)
treatment_sort: number(5)
tariff: [10..)

STRUCTURE medical treatment
patient_id: number(6)
treatment_code: number(4)
specialist_id: number(3)
assistent_id: number(3)
room_id: number(3)
date_of_treatment: DATE
duration: [1..)
fee: number(4)
success_factor: [0, 50, 100]

STRUCTURE medicines
medicine_code: char(6)

medicine_name:	char(20)
medicine_sort:	char(4)
danger_code:	[1..20]
unity:	[*GR*, *MG*, *ML*, *CC*, *ST*]

STRUCTURE med dispensation

patient_id:	number(6)
medicine_cc	char(6)
begin_date	DATE
duration	[1..)
freq	[1..6)
	[1..)
.ctor:	[0, 25, 50, 75, 100]

ase is automatically filled with data by using a random fact generator for DHIS, which emented in Prolog. For instance, it generates an arbitrary number of patients matching the ieral structure of patients as Prolog facts. Also, it generates patients between which some family relation exists. Four DHIS databases were generated, in which the number of facts vary from about 2500 to about 7500 facts in intermediate steps of appoximately 1500. Most of the facts concern patients facts and medicine dispensation facts. Because of the number of facts, as is only natural in real database applications, the data are stored on secondary storage. Hence, these Prolog facts were transformed in DBaseIV files, although they could be transformed to any database format, and saved on disk. However, in order to access these data by Prolog, Prolog is coupled to DBaseIV in the way described in CHAPTER 6.

The average access timings for accessing a fact from DBaseIV files may differ from the average access timings for accessing a fact from another database format file. For instance, managing data in ORACLE is handled substantially quicker than in DBaseIV. However, the number of accesses remains the same. Therefore, the number of accesses gives a good indication of the real benefit of a chosen method. The four databases are used to test some methods for integrity constraint checking and observe their behaviour, when the number of facts in the database vary.

A.2.3 Test Sets

In DHIS rules and integrity constraints are specified for adding more semantics to the database. Four sets are given. These sets of rules and constraints are used to test some methods for integrity constraint checking and observe their behaviour, when the number of rules and constraint vary (see A.2.4). The following sets are distinguished:

SET 1

family rules

IF X is child of Y
THEN Y is parent of X

IF X is parent of Y and Z
THEN Y is sibling of Z

IF X is female and parent of Y
THEN X is mother of Y

IF X is male and parent of Y
THEN X is father of Y

IF Y is male and sibling of X
THEN Y is brother of X

IF Y is female and sibling of X
THEN Y is sister of X

IF X and Y are married
THEN X is married_with Y

IF X and Y are married
THEN Y is married_with Y

IF X is female and married_with Y
THEN X is wife of Y

IF X is male and married_with Y
THEN X is husband of Y

IF X is parent of Y
THEN Y is descendant of X

IF Z is parent of X
AND Z is descendant of Y
THEN X is descendant of Y

IF X is descendant of Y
THEN X is relative of Y

IF X is descendant of Y
THEN Y is relative of Y

IF X and Y are descendants of Z
THEN X and relatives

IF X is child
AND sex of male
THEN X is of Y

IF X f Y
AND sex . is female
THEN X aughter of Y

tive of Y
nd Y are both male
THEN X is male_related to Y

IF X is relative of Y
AND X and Y are both female
THEN X is female_related to Y

IF X is male_related to Y
THEN X and Y are sex_related

IF X is female_related to Y
THEN X and Y are sex_related

family constraints

IF person X is father of Y
THEN X MUST be at least 17 years older than Y

IF person X is mother of Y
THEN X MUST be at least 15 years older than Y

PROHIBIT (person X is married to two persons Y and Z)

PROHIBIT (person X is married to Y who is relative of X)

PROHIBIT (X is child of itself)

IF X is a child
PROHIBIT (X is married)

PROHIBIT (the treatment of a patient X by a specialist Y if Y
treated a relative of X with success factor 0

elementary constraints

PROHIBIT (a date of registration before the date of birth)

IF X is a head of a department
THEN X MUST work full time

IF specialist X is replaced by specialist Y
THEN X and Y MUST have the same specialism

PROHIBIT (a specialist is replaced by him or herself)

PROHIBIT (the number of beds of a specialist who is working N hours
a week may not exceed N/2)

PROHIBIT (number of patients of a family doctor exceeding 200)

IF X is a family-doctor of Y
THEN X and Y MUST have the same residence

SET 2

age rules

IF a person X is less than one year old
THEN X is a baby

IF a person X is between 1 and 9 years old
THEN X is an infant

IF a person X is between 9 and 18 years old
THEN X is a teenager

IF a person X is older than 18 years old
THEN X is an adult

IF a person X is older than 64
THEN X is aged

IF X is a baby
THEN X is a child

IF X is an infant
THEN X is a child

IF X is a teenager
THEN X is a child

rules related to condition of patient

IF the age of a patient is 70 or older and sex is female
THEN the patient is weak

IF the age of a patient is 65 or older and sex is male
THEN the patient is weak

IF the illness is bad
THEN the patient is weak

IF e illness is hartattack
THEN the patient is weak

related to condition of patient

IF patient is pregnant
THEN patient MUST be female

IF patient is weak
THEN patient MUST have at most one room-mate

PROHIBIT (the dispensation of a strong medicine to a weak patient)

PROHIBIT (weak patient in department Q)

department constraints

IF treatment is pol inical
THEN patient MUS epart nt P

IF patient X has an illness that is genetic
THEN patient MUST be in department G

IF X is room in Q
THEN X MUST contain one bed

IF patient X has an illness that is infectious
THEN patient MUST be in department Q

IF room-type is laboratory, medical or for service
THEN number of beds MUST be 0

IF room-type is MED
THEN for each bed there MUST be at least 10 m^2 space

PROHIBIT (the occupation of two or more beds by one patient)

SET 3

medicine rules

IF danger code of medicine is 15-17
THEN medicine is strong

IF danger code of medicine is 18-20
THEN medicine is dangerous

IF danger code of medicine is greater than 13
THEN medicine is dangerous for pregnant women

IF medicine sort of a medicine is hormonal
THEN medicine is dangerous for pregnant women

constraints on medicine dispensation

IF danger code of medicine is below 10
THEN maximum frequency of this medicine is 5

IF danger code of medicine is between 9 and 16
THEN maximum frequency of this medicine is 3

IF danger code of medicine is greater than 15
THEN maximum frequency of this medicine is 1

PROHIBIT (two identical medicine dispensations to a patient)

IF reason of admission of a patient is pregnancy
THEN medicine dispensation for patient MUST have a maximum frequency of 2

PROHIBIT (the dispensation of a medicine for a pregnant patient
which is dangerous for pregnant women)

IF the medicine sort of a medicine is hormonal
THEN the danger code of the medicine MUST be greater than 10

PROHIBIT (dispensation of medicine of sort 'HOR' and
medicine of sort 'ANBC' to a patient)

PROHIBIT (dispensation of medicine 'SOMATONORM' and
medicine 'STOMBA' to a patient)

PROHIBIT (dispensation of medicine 'DURABOLIN' and
medicine 'HUMATROPE' to a patient)

PROHIBIT (dispensation of two strong medicines to a patient)

PROHIBIT (dispensation of dangerous medicines to a patient)

PROHIBIT (dispensation of two strong medicines to one patient)

PROHIBIT (the dispensation of a medicine for a second time to a patient for
the same illness while the first time was not successful, success factor 0).

SET 4:

illness rules

IF treatment_sort is '02','05','06','09','10','11','12','PC' or 'UP'
THEN treatment_name is an illness

IF X is relative of Y
AND X and Y are blood_identical
THEN X has blood relation with Y

IF X has same blood_type as Y
AND X has same rhesus factor as Y
THEN X is blood_identical to Y

IF X and Y are sex_related
AND patients X and Y have both illness A
THEN illness A is genetic for X

illness constraints

IF illness of patient X is genetic
AND X is sex_related to Y who had a medicine A with success factor below 50
AND X and Y have the same illness
THEN medicine A MUST not be dispensated to X

IF X is blood_related to Y who had medicine A with success factor below 75
THEN medicine A MUST not be dispensated to X

These integrity constraints must be obtained from experts in the hospital environment. Therefore, the knowledge engineer is responsible for getting all the integrity constraints in the hospital clear. Here, some documentation was used in order to find these rules and integrity constraints. Although some rules and/or integrity constraints may seem a little artificial, they give an idea how rules and constraints can be used in practice.

The rules can easily be expressed in Prolog. For instance, if the structure `family_relation` is represented in the database as a table `FAMILY` and the table is accessible via Prolog by the clause `fam(Pat,Rel,Fam)`, then the `child` resp. `married` relation can be built on top of this table by the following rules:

```
child(X,Y) :-
  fam(X,child,Y).

married(X,Y) :-
  fam(X,married,Y).
married(Y,X) :-
  fam(X,married,Y).
```

When we want to reason about a patient's age or sex, it is possible to extract the predicate `age` resp. `sex` from the table `PATIENT` by the following rules:

```
age(X,N) :-
  pat(X,_,_,_,_,_,_,_,DofB,_,_,_,_,_),
  age_of(DofB,N).
```

```
sex(X,male) :-
  pat(X,_,_,_,_,_,_,_,_,_,_,_,'M',_).
sex(X,female) :-
  pat(X,_,_,_,_,_,_,_,_,_,_,_,'F',_).
```

where pat is the predicate with thirteen arguments in Prolog, which accesses a PATIENT table with thirteen columns which in its turn is the database implementation of the structure patient, and where age_of is an auxiliary predicate (and therefore no part of the domain of the database), which delivers the age of a person when his date of birth is known. Other relations are defined, like parent, sibling, mother, father, relative, etc. in terms of existing relations. For instance:

```
parent(X,Y) :-
  child(Y,X).

sibling(Y,Z) :-
  parent(X,Y),
  parent(X,Z),
  not Y = Z.

mother(X,Y) :-
  sex(X,female),
  parent(X,Y).
```

The inconsistency rules can be expressed in Prolog easily. For instance, the constraints with respect to family are expressed in Prolog by:

```
ii(ii_id(1),[father(X,Y),age(X,N),age(Y,M),N < M + 17]).
ii(ii_id(2),[mother(X,Y),age(X,N),age(Y,M),N < M + 15]).
ii(ii_id(3),[married(X,Y),married(X,Z),not Y = Z]).
ii(ii_id(4),[married(X,Y),relative(Y,X)]).
ii(ii_id(5),[child(X,X)]).
ii(ii_id(6),[child(X,Y),married(X,Z)]).
ii(ii_id(7),[med_tr(X,_,Sp,_,_,_,_,_,_), relative(X,Y),
   med_tr(Y,_,Sp,_,_,_,_,_,0)]),
```

where med_tr is the predicate with nine arguments in Prolog, which accesses a MEDICAL TREATMENT table with nine columns which in its turn is the database implementation of the structure medical treatment. As we can see the domain part can be represented in Prolog in a nice and declarative way.

A.2.4 Tests and Results

The goal of the tests is to give insight in the performance of the new method when increasing the number of facts, rules and constraints. The following tests based on the test sets of A.2.3 were

performed.

Test 1 is intended to find out how the consistency checking methods for a particular database perform, when the number of rules and inconsistency indicators increases, for a varying number of facts. Four test sets of rules and inconsistency indicators were specified in A.2.3. In this test, the following transaction is used:

```
[[+,pat(963638, 'Robert', 'Streep', 'Kingsroad', 65, '6647 HL', 'Almere',
        '05/12/37', '28/11/93', 'B', +, m, 65) ],
 [+,pat(962638, 'Jane', 'Streep', 'Kingsroad', 65, '6647 HL', 'Almere',
        '04/10/38', '28/11/93', 'B', +, v, 65) ],
 [+,fam(963638,married,962638)],
 [+,med_tr(962638, 8110, 332, 124, 1533, '28/11/93', 40, 139, 100)],
 [+,med_tr(963638, 8110, 332, 124, 1533, '28/11/93', 40, 139, 75)],
 [+,dis(143208, 'ANHM02', '28/11/93', 1, 1, 1, 100)]],
```

where `dis` is the predicate with seven arguments in Prolog, which accesses a `DISPENSATION` table with seven columns which in its turn is the database implementation of the structure medicine dispensation. In test 1 the transaction does not influence the consistency of the database, although the transaction is relevant to several inconsistency indicators. We are interested in the timings of the check for this test for the test sets {1}, {1,2}, {1,2,3} and {1,2,3,4} respectivily. The results of test 1 can be found in Table A.1. When we look at these results some interesting issues come up. First note that the number of database accesses are a good indication for the performance, because an access to a database is far more expensive than any other computation to answer the query. These results show that the number of database accesses can be reduced considerably by the new method. Test set 4 is intended to show that the method based on induced updates performs weakly in some particular cases. A transaction which generates a lot of induced updates and a few potential updates, while only a few of these induced updates are necessary for the inconsistency check, cause a bad performance in the case of the method based on induced updates.

Consider for instance the following rule in test set 4:

IF X has same blood_type as Y

AND X has same rhesus factor as Y

THEN X is blood_identical to Y

Now, when we insert a new patient X, for whom the blood_type and rhesus factor are known, a lot of updates in the relation blood_identical are derivable. Each patient Y in the database with the same blood_type and rhesus factor will lead to an induced update in the relation blood_identical. Note that all these induced updates are subsumed by one potential update. Next consider the following rule in test set 4:

IF X is relative of Y

AND X and Y are blood_identical

Method based on Induced Updates	Number of Facts			
	2703	4284	5936	7425
Set 1	73.7	73.7	76.2	76.2
db-accesses	*86*	*86*	*86*	*86*
Set 1 and 2	103.9	105.1	107.3	109.5
db-accesses	*113*	*113*	*113*	*113*
Set 1, 2 and 3	130.5	133.3	133.9	132.9
db-accesses	*133*	*133*	*133*	*133*
Set 1, 2, 3 and 4	1920	3637	5283	6987
db-accesses	*1809*	*3379*	*4869*	*6391*

Method based on Potential Updates	Number of Facts			
	2703	4284	5936	7425
Set 1	54.5	57.8	58.8	58.2
db-accesses	*60*	*60*	*60*	*60*
Set 1 and 2	131.9	131.33	136.77	135.2
db-accesses	*116*	*116*	*116*	*116*
Set 1, 2 and 3	173.0	173.5	169.4	169.5
db-accesses	*147*	*147*	*147*	*147*
Set 1, 2, 3 and 4	342.1	352.2	348.8	353.0
db-accesses	*351*	*351*	*351*	*351*

Method based on Inconsistency rules	Number of Facts			
	2703	4284	5936	7425
Set 1	35.6	36.7	36.1	37.8
db-accesses	*39*	*39*	*39*	*39*
Set 1 and 2	63.5	65.2	64.5	68.2
db-accesses	*66*	*66*	*66*	*66*
Set 1, 2 and 3	98.4	97.2	102.2	101.3
db-accesses	*87*	*87*	*87*	*87*
Set 1, 2, 3 and 4	152.6	153.5	154.0	151.8
db-accesses	*151*	*151*	*151*	*151*

Table A.1: Timings of test1.

THEN X has blood relation with Y

When the relation "blood relation" appears in an inconsistency indicator, only those patients, who are blood_identical to X and are relatives of X, are needed for the consistency check. So, the generation of the relatives should have a higher priority than the generation of all blood_identical people. This might prevent the generation of an enormous number of blood_identical people of the inserted patient. This is not prevented in the method based on induced updates, because all induced updates are generated first. This explains the bad performance in the case of the methods based on induced updates, especially in the case of test set 4. So, one of the conclusions of the test is that the method based on induced updates has serious shortcomings.

Test 2 is intended to show how the method based on revised inconsistency rules performs compared to the other consistency checking methods in several particular cases. In test 2 the database is constrained to test set 1 only. Test 2 consists of three parts, with different interactions of the transaction with the database, namely

(i) a transaction may imply a lot of induced and potential updates, but none of these updates influences any indicator;

(ii) a transaction may imply a lot of induced and potential updates, influencing inconsistency indicators;

(iii) a transaction may imply a lot of potential updates and only a few induced updates, where some of the potential updates appear in inconsistency indicators, while they do not subsume any induced update.

In the first two cases we can recognize redundancy of the first type and in the third case we can recognize redundancy of the second type; both are described in CHAPTER 3.

In the first part of test 2, we take test set 1 of the rules and inconsistency indicators, but with the family constraints skipped. Further, suppose we have the following transaction for this part of test 2.

```
[[+,pat(963638, 'Robert' ,'Streep', 'Kingsroad', 65, '6647 HL', 'Almere',
        '05/12/68', '28/11/93', 'B', +, m, 65)],
 [+,pat(962638, 'Jane', 'Streep', 'Kingsroad', 65, '6647 HL', 'Almere',
        '04/10/38', '28/11/93', 'B', +, v, 65)],
 [+,fam(963638,child,962638)],
 [+,med_tr(962638, 8110, 332, 124, 1533, '28/11/93', 40, 139, 100)],
 [+,med_tr(963638, 8110, 332, 124, 1533, '28/11/93', 40, 139,  75)],
 [+,dis(143208, 'ANHM02', '28/11/93', 1, 1, 1, 100)]].
```

Note the difference between this transaction and the transaction of test 1. This transaction contains the insertion of a family-fact, which implies a great number of updates caused by the application of several database rules for family relations by using several family-facts of relatives of each of

the patients involved in the transaction that are stored in the database. Hence, this transaction will imply a large number of induced and potential updates, expressing the new knowledge about these members. However, because in this case no inconsistency indicators exist concerning the family relations, none of the induced resp. potential updates leads to an inconsistency check. In the proposed method this will be noted immediately, because no inconsistency rules are triggered by the update. However, in the other methods a lot of preprocessing has to be done. Before finding the inconsistency indicators that must be checked, we have to compute the induced updates resp. potential updates. Therefore, in this case the method based on inconsistency rules will perform superior compared to the other methods, as the results in Table A.2 show.

Method based on	**timing**	**db-accesses**
induced updates	692.1	513 (41)
potential updates	14.1	0 (143)
inconsistency rules	0.05	0

Table A.2: Timings for part one of test 2

In the tables of test 2, the number of induced updates resp. potential updates are given in brackets in the column of db-accesses. In the potential update method and in the method based on inconsistency rules there are no database accesses needed, while the induced update method must access the database 513 times in order to find the induced updates. In this case all accesses were redundant, because none of the induced updates influences any indicator. Although no database accesses are needed in the potential update method, a lot of time is wasted in order to derive the 143 potential updates.

In the second part of test 2, test set 1 of the rules and inconsistency indicators is used as the intensional database. So, compared to the previous part we now do have constraints on family relations. The same transaction as in the previous test is used, which, as we have stated before, implies a great number of induced resp. potential updates. However, in this test there exists a large number of induced and potential updates that lead to checks of several inconsistency indicators. The results of this part of test 2 can be found in Table A.3. Note the asterisk $*$ in the last two columns. Because of the presence of the recursive relation `relative` the update in `fam` will always lead to a potential update `relative(X,Y)`, where X and Y are variables. So, each indicator containing `relative(X,Y)` has to be fully checked. For instance, in this case the update together with the two indicators:

```
ii(ii_id(4),[married(X,Y),relative(Y,X)]).
ii(ii_id(7),[med_tr(X,_,Sp,_,_,_,_,_,_), relative(X,Y),
    med_tr(Y,_,Sp,_,_,_,_,_,0)]).
```

causes a full database search of the relations `married` and `med_tr` (medical treatment). For each success the resulting instance of the relation `relative` has to be evaluated. The results in the table

marked with an asterisk are corresponding to the same test, but without the inconsistency indicators 4 and 7. In this case, these inconsistency indicators are left out. Compared to part one of

Method based on	timing	db-accesses	timing *	db-accesses *
induced updates	763.4	703 (41)	719.6	593 (41)
potential updates	>1 hour	>30000 (143)	43.6	22 (143)
inconsistency rules	42.9	44	18.5	12

Table A.3: Timings for part two of test 2

test 2, the number of database accesses increases in all methods, because now also several inconsistency indicators have to be checked, for which the database has to be accessed several times. In the method based on potential updates this causes the explosive growth of the number of database accesses: the number of database accesses exceeds the number of 30000. It is obvious that the method based on potential updates will lose its efficiency, when recursive predicates appear in the inconsistency indicators.

In the third part of test 2, test set 1 of the rules and inconsistency indicators is used as the intensional database. Here, the transaction is an insertion of `[+,fam(963738,child,624213)]`. For this transaction only a few induced updates are derivable, while a lot of potential updates are derivable because of the large number of rules depending on the `fam`-relation. The results of this part of test 2 can be found in Table A.4. It shows the performance of each method, when only a few induced updates (12) and a lot of potential updates (136) exist. In this case, the test is divided in a test that includes the inconsistency indicators 4 and 7 and a test (indicated by $*$) that excludes these inconsistency indicators. Obviously, in the case of a few induced updates the method of induced updates will perform better than in the other tests. However, even in this situation the method based on inconsistency rules performs better. It is also clear from the results that the method based on potential updates will perform well as long as there are no recursive rules in the indicators. Note that, in this appendix, some tests were performed on a deductive database, where

Method based on	timing	db-accesses	timing *	db-accesses *
induced updates	58.3	36 (12)	49.8	29 (12)
potential updates	>1 hour	>30000 (136)	26.8	6 (136)
inconsistency rules	27.3	25	10.4	5

Table A.4: Timings for part three of test 2

the data are stored on secondary memory. This means that the access timings increase compared to a deductive database, where the data are available in the main memory of Prolog. In [Sel97] some tests were performed with main memory databases. Further, note that the number of database accesses can be reduced in all methods, when we implement DHIS by using a real deductive

database management system, which allows the optimization of a set of queries. Because the revised inconsistency rules are known at compile time, they are suitable for optimization at compile time.

References

[Bro78] M. BRODIE. *Specification and Verification of Database Semantic Integrity.* PhD thesis, University of Toronto, 1978.

[Sel93] RON R. SELJÉE. Expert Database Systems with Integrity Constraint Checking. Technical report, pages 1–28. Centre for Knowledge Engineering (CIBIT), Utrecht, The Netherlands, 1993.

[Sel94] RON R. SELJÉE. Expert Database Systems with Integrity Constraint Checking; APPENDIX. Technical report, pages 29–55. Centre for Knowledge Engineering (CIBIT), Utrecht, The Netherlands, 1994.

[Sel97] RON R. SELJÉE. *FICCS; An Integrity Constraint Checking System for the Validation of Semantic Integrity Constraints in Consistent Deductive Databases.* PhD thesis, Co-operation Centre of Tilburg and Eindhoven Universities, Tilburg, The Netherlands, 1997.

Index

Curriculum Vitae

Ron Seljée studied Mathematics at the University of Amsterdam. He graduated, with a specialisation in Probabilistics, in 1989. With this graduation he also obtained a first-degree teacher's license. In 1990 he started a part-time Master of Science Course in "Knowledge Engineering" at the Center for Knowledge Engineering in Utrecht. Later that year he became a researcher at the University of Tilburg at the Department of Philosophy in order to propose a research project in the field of deductive databases. In 1991 he started his proposed project *Integrity Constraint Checking in Deductive Databases* as a research assistent at the Cooperation Center of the Tilburg and Eindhoven Universities, stationed at the Department of Philosophy of Tilburg University and the Department of Mathematics and Computing Science of Eindhoven University of Technology respectivily. The results of this project can be found in his dissertation, in which you are reading now. In 1994 he received his M. Sc. degree from Utrecht. As the final project of the Master of Science course he implemented an experimental Deductive Hospital Information System (DHIS) with an automated integrity constraint checker.

Acknowledgements

I would like to thank the people who contributed directly to the realization of this thesis. First of all, I would like to thank Harrie de Swart for his confidence and support during all those years of research. Despite his own numerous activities, he always found the time to discuss minor and major issues concerning my research. Without his confidence and support this thesis could and would not have been written. Also, I would like to thank Paul de Bra of the university of Eindhoven for his comments on the preliminary versions of this thesis and for showing his confidence by accepting the promotorship at a later stage. Further, despite their anonymity, I would like to thank all referees of my papers. Their comments were also very helpful.

Further, I want to thank the co-operation Centre of the Universities Tilburg and Eindhoven and their co-workers for their co-operation and the faculty of philosophy for their hospitality and co-operation, especially the lady secretaries, who allowed me to drive them from their PC in order to get a print-job to the one and only postscript printer, often just before closing time, which may have led to some inconvenience once in a while.

And last but not least, I would like to thank all people who made my stay in Tilburg a pleasant one. Besides my neighbour and promotor Harrie de Swart, I would like to thank my room-mates during all those years for their flexible way they shared room P3.210; Eric "Cajun" de Kogel, who was my room-mate for most of my stay in Tilburg and whose leave could only be filled up by employing three research assistents; Twan Laan, who is probably sitting in a train right now, which obviously will lead to a lot of inconvenience; Bart "Low Noise" Knaack, who shows that besides himself his laptop is completely multimedia; and Michael Franssen, whose favourite film is, if I remember it right, "The postman always ray traces twice".

Other colleagues, who I want to acknowledge, are Willy Ophelders and Ton Storcken for their invitations for, and advises and conversations during, their coffee-breaks in the first two years of my research; and, especially, Richard Starmans, Robbert van Baaren and once again Eric de Kogel (and not forgetting all people who happened to join us occasionally) are acknowledged for their conversations during lunch-time and coffee-breaks, which were an indispensable distraction of my daily mental activities.

Samenvatting

Aan informatiesystemen worden steeds hogere eisen gesteld. Voorheen was het de verantwoordelijkheid van de gebruiker om zelf de gegevens opgeslagen in de database te interpreteren en om te zetten in bruikbare informatie. Nu en in de toekomst wordt de interpretatie van de data meer en meer bij het systeem gelegd. Nieuwe data modellen en bijbehorende inferentie mechanismen worden ingezet om systemen in staat te stellen de informatie op een gestructureerde wijze aan de gebruiker aan te bieden. Een voorbeeld systemen, die dit beogen te bereiken, zijn deductieve database systemen. In deze systemen kan door middel van deductieregels informatie afgeleid worden uit de data, die voorgesteld kunnen worden als opgeslagen in de tabellen uit het relationele model.

Verder worden er aan informatiesystemen vaak integriteitscondities toegevoegd, om aan te geven waaraan de opgeslagen informatie moet voldoen. Dit om een goede weergave van de werkelijkheid, die het informatiesysteem modelleert, te garanderen. Een informatiesysteem heet consistent indien het informatiesyteem aan al zijn integriteitscondities voldoet. In het algemeen wordt, bij het bestuderen van methoden om integriteitscondities te controleren, uitgegaan van consistente informatiesystemen. Integriteitscondities worden pas gecontroleerd indien een consistent informatiesysteem wordt geactualiseerd. Dit heeft het voordeel dat de oorzaak van een eventuele inconsistentie via de update gevonden kan worden. Daardoor kan het zoekproces naar een inconsistentie een stuk efficiënter worden.

In dit proefschrift wordt het systeem *FICCS* beschreven, dat integriteitscondities, uitgedrukt in de eerste orde predicaten logica, in gestratificeerde deductieve database systemen automatisch controleert. Hierbij gaat het niet om het herstellen van de consistentie na een transactie, maar om een eventuele inconsistentie zo snel mogelijk te ontdekken.

Daarby wordt gebruik gemaakt van inconsistentieregels, die – door updates uit een transactie erop toe te passen – een mogelijke inconsistentie opsporen. Inconsistentieregels worden geconstrueerd uit de verzameling deductieregels en integriteitscondities, onafhankelijk van de feiten uit de database of feiten uit de transactie. Deze inconsistentieregels worden dan aan de deductieve database toegevoegd. Zolang er geen update plaats vindt van deductieregels of integriteitscondities, kunnen deze opnieuw gebruikt worden om de consistentie van een deductieve database te controleren na een transactie bestaande uit feiten. Een verandering in de verzameling deductieregels of integriteitscondities heeft een incrementele aanpassing van de verzameling inconsistentieregels tot gevolg. Derhalve hoeven na zo'n verandering niet al deze regels opnieuw geconstrueerd te worden.

Het proefschrift beschrijft eveneens de mogelijke redundante evaluaties die kunnen ontstaan bij methoden voor het controleren van integriteitscondities. Deze redundanties worden ingedeeld in soorten, elk met een eigen karakter. Het wordt aan de hand van deze klassificatie duidelijk waarom de op inconsistentieregels gebaseerde methode zo efficiënt is.

Het proefschrift laat bovendien zien dat in de gepresenteerde methode op een natuurlijke en efficiënte wijze negatie, recursie, existentiële quantoren en algemenere updates naast toevoegingen en verwijderingen, zoals vervangingen, opgenomen kunnen worden.

Tevens wordt de architectuur en de implementatie van *FICCS* gegeven. Het toont de elegante wijze waarop de methode te implementeren is; hier in de logische programmeertaal Prolog. Naast deze methode werden nog enkele andere methoden geïmplementeerd, die aan de hand van efficiëntietests op een fictieve deductieve ziekenhuisdatabase met elkaar werden vergeleken. Deze testresultaten laten de significante verbeteringen zien, die mogelijk zijn door gebruik te maken inconsistentieregels.

Het proefschrift is een van de resultaten van het onderzoeksproject 91P van het Samenwerkings-Orgaan Brabantse Universiteiten (SOBU).